Franck van Breugel
Marsha Chechik (Eds.)

LNCS 5201

CONCUR 2008 – Concurrency Theory

19th International Conference, CONCUR 2008
Toronto, Canada, August 2008
Proceedings

Springer

Lecture Notes in Computer Science 5201

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

Franck van Breugel Marsha Chechik (Eds.)

CONCUR 2008 - Concurrency Theory

19th International Conference, CONCUR 2008,
Toronto, Canada, August 19-22, 2008. Proceedings

Springer

Volume Editors

Franck van Breugel
York University, Department of Computer Science and Engineering, 4700 Keele Street, Toronto, M3J 1P3, Canada
E-mail: franck@cse.yorku.ca
Marsha Chechik
University of Toronto, Department of Computer Science, 10 King's College Road, Toronto, ON M5S 3G4, Canada
E-mail: chechik@cs.toronto.edu

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.1.3, D.3, F.1.2, C.2

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

ISSN 0302-9743
ISBN-10 3-540-85360-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-85360-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12450923 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 19th International Conference on Concurrency Theory (CONCUR 2008) which took place at the University of Toronto in Toronto, Canada, August 19–22, 2008. CONCUR 2008 was co-located with the 27th Annual ACM SIGACT-SIGOPS Symposium on the Principles of Distributed Computing (PODC 2008), and the two conferences shared two invited speakers, some social events, and a symposium celebrating the lifelong research contributions of Nancy Lynch.

The purpose of the CONCUR conferences is to bring together researchers, developers, and students in order to advance the theory of concurrency and promote its applications. Interest in this topic is continuously growing, as a consequence of the importance and ubiquity of concurrent systems and their applications, and of the scientific relevance of their foundations. Topics include basic models of concurrency (such as abstract machines, domain theoretic models, game theoretic models, process algebras, and Petri nets), logics for concurrency (such as modal logics, temporal logics and resource logics), models of specialized systems (such as biology-inspired systems, circuits, hybrid systems, mobile systems, multi-core processors, probabilistic systems, real-time systems, synchronous systems, and Web services), verification and analysis techniques for concurrent systems (such as abstract interpretation, atomicity checking, model-checking, race detection, run-time verification, state-space exploration, static analysis, synthesis, testing, theorem proving and type systems), and related programming models (such as distributed or object-oriented).

Of the 120 regular and 5 tool papers submitted this year, 33 regular and 2 tool papers were accepted for presentation and are included in the present volume. During the reviewing process, at least three reviews were collected for each regular paper and at least four reviews for each tool paper. In total, 416 reviews were collected.

The conference also included talks by:

- Tevfik Bultan (University of California, Santa Barbara, USA)
- Joseph Halpern (Cornell University, Ithaca, USA) – shared with PODC
- Prakash Panangaden (McGill University, Montreal, Canada) – shared with PODC
- Shaz Qadeer (Microsoft Research, Redmond, USA)

Abstracts of these talks can be found in the present volume.

The symposium "Nancy Lynch Celebration: Sixty and Beyond" included talks by:

- Hagit Attiya (Technion, Haifa, Israel)
- Michael Fischer (Yale University, New Haven, USA)
- Seth Gilbert (EPFL, Lausanne, Switzerland)

- Maurice Herlihy (Brown University, Providence, USA)
- Roberto Segala (University of Verona, Italy)
- Jennifer Welch (Texas A&M University, College Station, USA)

CONCUR 2008 had eight satellite workshops:

- Workshop on Approximate Behavioral Equivalences (ABE 2008), organized by Franck van Breugel
- International Workshop on Concurrency in Enterprise Systems (COINES 2008), organized by Matthias Anlauff and Asuman Suenbuel
- Workshop on Distributed Computing, Concurrency Theory, and Verification (DisCoVeri 2), organized by Yoram Moses, Uwe Nestmann and Mark R. Tuttle
- 15th International Workshop on Expressiveness in Concurrency (EXPRESS 2008), organized by Daniele Gorla and Thomas Hildebrandt
- Workshop on Formal Methods for Wireless Systems (FMWS 2008), organized by Jens Chr. Godskesen and Massimo Merro
- 10th International Workshop on Verification of Infinite-State Systems (INFINITY 2008), organized by Peter Habermehl and Tomás Vojnar
- 6th International Workshop on Security Issues in Concurrency (SecCo 2008), organized by Steve Kremer and Prakash Panangaden
- Young Researchers Workshop on Concurrency Theory (YR-CONCUR 2008), organized by Joost-Pieter Katoen and P. Madhusudan

We would like to thank the Program Committee members and the referees who assisted in the process of putting together an excellent scientific program for CONCUR. Many thanks to the Workshops Chair, Richard Trefler, and the workshop organizers. We would also like to thank the invited speakers, the authors of submitted papers, and the participants of the conference. Finally, we thank Joan Allen and many volunteers from the University of Toronto and York University without whom the conference could not run, as well as the organizers of PODC (Rida Bazzi, Faith Ellen, Boaz Patt-Shamir, Eric Ruppert, and Srikanta Tirthapura) and the organizers of the symposium for Nancy Lynch (Hagit Attiya, Victor Luchangco, Roberto Segala, Frits Vaandrager and Jennifer Welch), who helped shape the common agenda of the event.

We gratefully acknowledge the use of easychair for conducting the review process and support from ACM's SIGACT and SIGOPS, the Fields Institute, IBM, Microsoft, SAP, the Department of Computer Science of the University of Toronto, and the Department of Computer Science and Engineering of York University.

June 2008

Franck van Breugel
Marsha Chechik

Organization

General Chairs

Franck van Breugel	York University, Canada
Marsha Chechik	University of Toronto, Canada

Workshop Chair

Richard Trefler	University of Waterloo, Canada

Steering Committee

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

Program Committee

Luca de Alfaro	University of California, Santa Cruz, USA
Pedro R. D'Argenio	Universidad Nacional de Cordoba, Argentina
Jos Baeten	Eindhoven University of Technology, The Netherlands
Christel Baier	Technical University Dresden, Germany
Eike Best	Carl von Ossietzky Universität Oldenburg, Germany
Dirk Beyer	Simon Fraser University, Canada
Patricia Bouyer	LSV, CNRS & ENS Cachan, France
Mario Bravetti	University of Bologna, Italy
Franck van Breugel	York University, Canada
Ilaria Castellani	INRIA Sophia Antipolis, France
Marsha Chechik	University of Toronto, Canada
Wan Fokkink	Vrije Universiteit Amsterdam/CWI, The Netherlands
Rob van Glabbeek	National ICT, Australia
Arie Gurfinkel	Carnegie Mellon University, USA
Anna Ingolfsdottir	Reykjavik University, Iceland
Radha Jagadeesan	DePaul University, USA
Barbara König	University of Duisburg-Essen, Germany

Marta Kwiatkowska	University of Oxford, UK
Orna Kupferman	Hebrew University, Israel
Kim Larsen	Aalborg University, Denmark
Nancy Lynch	MIT, USA
P. Madhusudan	UIUC, USA
Ugo Montanari	University of Pisa, Italy
Anca Muscholl	Université Bordeaux, France
Catuscia Palamidessi	INRIA Futurs and LIX, France
Corina Pasareanu	Perot Systems/NASA Ames Research Center, USA
Scott Smolka	SUNY at Stony Brook, USA
Nobuko Yoshida	Imperial College London, UK

Referees

Parosh Abdulla
Lucia Acciai
Luca Aceto
Alex Aiken
Nina Amla
Miguel Andres
Jesus Aranda
Eric Badouel
Paolo Baldan
Damian Barsotti
Ezio Bartocci
Marek Bednarczyk
Emmanuel Beffara
Laurent Van Begin
Shoham Ben-David
Martin Berger
Jan Bergstra
Marco Bernardo
Nathalie Bertrand
Bruno Blanchet
Roderick Bloem
Mihaela Bobaru
Benedikt Bollig
Michele Boreale
Dragan Bosnacki
Gerard Boudol
Roberto Bruni
Doina Bucur
Mikkel Bundgaard
Marzia Buscemi
Luís Caires
Silvio Capobianco
Marco Carbone
Franck Cassez
Gian Luca Cattani
Rohit Chadha
Sagar Chaki
Krishnendu Chatterjee
Swarat Chaudhuri
Taolue Chen
Renato Cherini
Hana Chockler
Frank Ciesinski
Federica Ciocchetta
Rance Cleaveland
Thomas Colcombet
Ricardo Corin
Pieter Cuijpers
Ellie D'Hondt
Philippe Darondeau
Alexandre David
Yuxin Deng
Malte Diehl
Laurent Doyen
Kai Engelhardt
Marco Faella
Ulrich Fahrenberg
Azadeh Farzan
Xinyu Feng
Jerome Feret
Hans Fleischhack
Matthew Fluet
Riccardo Focardi
David de Frutos-Escrig
Fabio Gadducci
Diego Garbervetsky
Simon Gay
Blaise Genest
Sonja Georgievska
Naghmeh Ghafari
Fatemeh Ghassemi
Giorgio Ghelli
Giuseppe De Giacomo
Hugo Gimbert
Sergio Giro
Jens Chr. Godskesen
Valentin Goranko
Andy Gordon
Marcus Groesser
Dan Grossman
Elsa Gunter
Ichiro Hasuo
Reiko Heckel
Tobias Heindel
Maurice Herlihy
Thomas Hildebrandt
Martin Hilscher
Daniel Hirschkoff
Dennis Hofheinz
Ross Horne
Hans Hüttel
Selma Ikiz
Michael Johnson

Kristján Jónsson
Ozan Kahramanogullari
Elham Kashefi
Ekkart Kindler
Joachim Klein
Sascha Klüppelholz
Simon Kramer
Pavel Krcal
K. Narayan Kumar
Alberto Lluch Lafuente
Robby Lampert
Ivan Lanese
Rom Langerak
Francois Laroussinie
Slawomir Lasota
Luciano Lavagno
Matías D. Lee
Marina Lenisa
Martin Leucker
Shuhao Li
Cong Liu
Christof Loeding
Etienne Lozes
Victor Luchangco
Yoad Lustig
Bas Luttik
Pablo E. Martínez López
Markus Müller-Olm
Jan-Willem Maessen
Matteo Maffei
Sergio Maffeis
Louis Mandel
Nicolas Markey
Tomasz Mazur
Hernan Melgratti
Massimo Merro
Roland Meyer
Marino Miculan
Bojanczyk Mikolaj
Paul Miner
Arjan Mooij
Patrice Moreaux
Dimitris Mostrous
M.R. Mousavi
Madhavan Mukund
Francesco Zappa Nardelli
Uwe Nestmann
Mogens Nielsen
Gethin Norman
Ulrik Nyman
Carlos Olarte
Simona Orzan
Rotem Oshman
Joel Ouaknine
Luca Padovani
David Parker
Gennaro Parlato
Joachim Parrow
Michael D. Pedersen
Jorge A. Perez
Gustavo Petri
Corin Pitcher
Nir Piterman
Damien Pous
Pritam Roy
Astrid Rakow
C.R. Ramakrishnan
Julian Rathke
Stefan Ratschan
Antonio Ravara
Anders Ravn
Michel Reniers
Arend Rensink
John Reppy
Tamara Rezk
Grigore Rosu
Diptikalyan Saha
Arnaud Sangnier
Stefan Schwoon
Roberto Segala
Olivier Serre
Anu Singh
Arne Skou
Pawel Sobocinski
Jeremy Sproston
Jiri Srba
Sam Staton
Kohei Suenaga
Murali Talupur
Olivier Tardieu
Anya Tefliovich
Peter Thiemann
Paul van Tilburg
Simone Tini
Salvatore La Torre
Angelo Troina
Harvey Tuch
Andrea Turrini
Sebastian Uchitel
Irek Ulidowski
Frank Valencia
Margus Veanes
Björn Victor
Hugo Vieira
Maria Grazia Vigliotti
Jules Villard
Erik de Vink
Mahesh Viswanathan
Walter Vogler
Marc Voorhoeve
Bjoern Wachter
Igor Walukiewicz
Bow-Yaw Wang
Muck van Weerdenburg
Ou Wei
Gera Weiss
Elke Wilkeit
Glynn Winskel
Verena Wolf
James Worrell
Jaap van der Woude
Shoji Yuen
Gianluigi Zavattaro
Lijun Zhang
Roberto Zunino

Sponsors

ACM's SIGACT and SIGOPS
Department of Computer Science, University of Toronto
Department of Computer Science and Engineering, York University
Fields Institute, Toronto
IBM
Microsoft
SAP

Table of Contents

Invited Papers

Contributed Papers

Beyond Nash Equilibrium: Solution Concepts for the 21st Century

Joseph Y. Halpern*

Cornell University

Nash equilibrium is the most commonly-used notion of equilibrium in game theory. However, it suffers from numerous problems. Some are well known in the game theory community; for example, the Nash equilibrium of repeated prisoner's dilemma is neither normatively nor descriptively reasonable. However, new problems arise when considering Nash equilibrium from a computer science perspective: for example, Nash equilibrium is not robust (it does not tolerate "faulty" or "unexpected" behavior), it does not deal with coalitions, it does not take computation cost into account, and it does not deal with cases where players are not aware of all aspects of the game. Solution concepts that try to address these shortcomings of Nash equilibrium are discussed.

The paper appears in the *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing*, 2008.

* Work supported in part by NSF under grants ITR-0325453 and IIS-0534064, and by AFOSR under grant FA9550-05-1-0055.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, p. 1, 2008.

Service Choreography and Orchestration with Conversations

Tevfik Bultan

Department of Computer Science
University of California, Santa Barbara
bultan@cs.ucsb.edu

Service oriented computing provides technologies that enable multiple organizations to integrate their businesses over the Internet. Typical execution behavior in this type of distributed systems involves a set of autonomous peers interacting with each other through messages. Modeling and analysis of interactions among the peers is a crucial problem in this domain due to following reasons: 1) Organizations may not want to share the internal details of the services they provide to other organizations. In order to achieve decoupling among different peers, it is necessary to specify the interactions among different services without referring to the details of their local implementations. 2) Modeling and analyzing the global behavior of this type of distributed systems is particularly challenging since no single party has access to the internal states of all the participating peers. Desired behaviors have to be specified as constraints on the interactions among different peers since the interactions are the only observable global behavior. Moreover, for this type of distributed systems, it might be worthwhile to specify the interactions among different peers before the services are implemented. Such a top-down design strategy may help different organizations to better coordinate their development efforts.

This type of distributed systems can be modeled as a composite Web service that consists of a set of peers that interact with each other via synchronous and/or asynchronous messages [3]. A conversation is the global sequence of messages exchanged among the peers participating in a composite Web service [2]. A choreography specification identifies the set of allowable conversations for a composite Web service. An orchestration, on the other hand, is an executable specification that identifies the steps of execution for the peers.

This conversation based modeling framework leads to the following interesting problems: realizability, synthesis, conformance, and synchronizability [4]. A choreography specification is realizable if the corresponding conversation set can be generated by a set of peers [6,9]. This step is necessary to guarantee that the choreography specifications that are developed in a top-down manner are implementable. A related problem is automated synthesis of peer implementations from a given choreography specification. Another interesting problem is investigating the conformance between orchestration and choreography specifications. An orchestration specification conforms to a choreography specification if the global sequence of messages generated by the orchestration is allowed by the choreography specification. Finally, synchronizability analysis [5,10] investigates the effects of

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 2–3, 2008.

asynchronous versus synchronous communication to improve the efficiency of interaction analysis. A set of asynchronously communicating peers are synchronizable if their conversation set does not change when asynchronous communication is replaced with synchronous communication. Replacing asynchronous communication with synchronous communication enables more efficient analysis by removing the communication channels from the state space of the system.

Web Service Analysis Tool (WSAT) [7] is a tool for analyzing conversations. WSAT verifies LTL properties of conversations, checks sufficient conditions for realizability and synchronizability and synthesizes peer implementations from choreography specifications. In order to model XML data, WSAT uses a guarded automata model where the guards of the transitions are written as XPath expressions. WSAT uses the explicit-state model checker SPIN [11] for LTL model checking by translating the guarded automata model to Promela [8]. WSAT has been used to analyze realizability and synchronizability of composite Web services specified using the BPEL orchestration language [1] and conversation protocols [6], which is a formalism for choreography specification and analysis.

References

1. Web services business process execution language version 2.0 (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
2. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: A new approach to design and analysis of e-service composition. In: Proceedings of the Twelfth International World Wide Web Conference, pp. 403–410 (May 2003)
3. Bultan, T., Fu, X., Su, J.: Analyzing conversations of web services. IEEE Internet Computing 10(1), 18–25 (2006)
4. Bultan, T., Fu, X., Su, J.: Analyzing conversations: Realizability, synchronizability, and verification. In: Baresi, L., Di Nitto, E. (eds.) Test and Analysis of Web Services, pp. 57–85. Springer, Heidelberg (2007)
5. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: Proceedings of the 13th International World Wide Web Conference, pp. 621–630 (May 2004)
6. Fu, X., Bultan, T., Su, J.: Conversation protocols: A formalism for specification and verification of reactive electronic services. Theoretical Computer Science 328(1–2), 19–37 (2004)
7. Fu, X., Bultan, T., Su, J.: WSAT: A tool for formal analysis of web services. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 510–514. Springer, Heidelberg (2004)
8. Fu, X., Bultan, T., Su, J.: Model checking XML manipulating software. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, pp. 252–262 (July 2004)
9. Fu, X., Bultan, T., Su, J.: Realizability of conversation protocols with message contents. International Journal of Web Services Research 2, 68–93 (2005)
10. Fu, X., Bultan, T., Su, J.: Synchronizability of conversations among web services. IEEE Trans. Software Eng. 31(12), 1042–1055 (2005)
11. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Eng. 23(5), 279–295 (1997)

Knowledge and Information in Probabilistic Systems

Prakash Panangaden

School of Computer Science, McGill University, Montreal, Quebec, Canada

Concurrency theory is in an exciting period of confusion. Confusion is always exciting because it heralds the coming of new ideas and deeper understanding. There are several ingredients in the cauldron: some new, some not so new. The old ingredients are process algebra, bisimulation and other equivalences and modal logics. The not-so-old ingredients are probability, mobility and real-time, and the new ingredients are knowledge, games and information theory.

Of course, knowledge in the form of epistemic logic – and especially common knowledge and it variants – is well known to the PODC community. It is well over twenty years since Halpern and Moses wrote their influential paper on common knowledge. Probabilistic epistemic logic and dynamic epistemic logic have also been extensively studied. It is time to synthesize these ideas with the world of concurrency theory.

How does concurrency theory accommodate the concept of knowledge? One way is to think of concurrent processes as agents playing games. We have just begun to explore these ideas. There are certain situations – arising in security – where epistemic concepts fit perfectly with the idea of agents playing games. The time is ripe for exploring new forms of process algebra inspired by the idea of processes being agents playing games. It is likely possible that this will lead to new types of interactions other than synchronization and value passing. Indeed, one can argue that composition of strategies is already an example of a new type of interaction between processes.

It is even possible that mobility can be understood as an epistemic concept: but now I am speculating wildly. Roughly speaking, the thinking is that spatial concepts underlie many recent developments in concurrency: for example, ambients. In many instances, for example, knowledge in distributed systems, the epistemic modality captures a local versus global view of space. Of course, much needs to be thought through.

Where does information theory fit in? If we are to think of knowledge as flowing between agents in a probabilistic system then it is natural to think of quantifying the "amount" of knowledge: this leads directly to ideas of information theory. I will describe some recent work along these lines by Parikh and his co-workers. Indeed, information theory may serve as a kind of probabilistic epistemic "logic" just as measure theory serves as a kind of probabilistic propositional logic: an analogy emphasized by Kozen.

I would like to thank Samson Abramsky for numerous enlightening conversations and for his inspirational paper, "Retracing some path in process algebra," an invited talk at CONCUR in 1996. I have also enjoyed enlightening conversations with Kostas Chatzikokolakis, Joe Halpern, Sophia Knight, Radha Jagadeesan, Catuscia Palamidessi, Rohit Parikh and especially Dexter Kozen who set me on the probabilistic path over 20 years ago.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, p. 4, 2008.

Taming Concurrency: A Program Verification Perspective

Shaz Qadeer

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA

Concurrency, as a basic primitive for software construction, is more relevant today than ever before, primarily due to the multi-core revolution. General-purpose software applications must find ways to exploit concurrency explicitly in order to take advantage of multiple cores. However, experience has shown that explicitly parallel programs are difficult to get right. To deliver compelling software products in the multi-core era, we must improve our ability to reason about concurrency.

Generalizing well-known sequential reasoning techniques to concurrent programs is fundamentally difficult because of the possibility of interference among concurrently-executing tasks. In this lecture, I will present *reduction* and *context-bounding* – two ideas that alleviate the difficulty of reasoning about interference by creating a simpler view of a concurrent execution than an interleaving of thread instructions. Reduction reduces interference by making a sequence of instructions in a thread appear as a single atomic instruction; it allows the programmer to view an execution as a sequence of large atomic instructions. Context-bounding reduces interference by focusing on executions with a small number of context switches; it allows the programmer to view an execution as a sequence of large thread contexts. I will present the theory behind these two approaches and their realization in a number of program verification tools I have worked on over the years.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, p. 5, 2008.

A Model of Dynamic Separation for Transactional Memory

Martín Abadi[1,2], Tim Harris[1], and Katherine F. Moore[1,3]

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

Abstract. Dynamic separation is a new programming discipline for systems with transactional memory. We study it formally in the setting of a small calculus with transactions. We provide a precise formulation of dynamic separation and compare it with other programming disciplines. Furthermore, exploiting dynamic separation, we investigate some possible implementations of the calculus and we establish their correctness.

1 Introduction

Several designs and systems based on transactions aim to facilitate the writing of concurrent programs. In particular, software transactional memory (STM) appears as an intriguing alternative to locks and the related machinery for shared-memory concurrency [14]. STM implementations often allow transactions to execute in parallel, optimistically, detecting and resolving conflicts between transactions when they occur. Such implementations guarantee that transactions appear atomic with respect to other transactions, but not with respected to direct, non-transactional accesses to memory. This property has been termed "weak atomicity" [6], in contrast with the "strong atomicity" that programmers seem to expect, but which can be more challenging to provide.

Therefore, it is attractive to investigate programming disciplines under which the problematic discrepancy between "weak" implementations and "strong" semantics does not arise. In these disciplines, basically, transactional and non-transactional memory accesses should not be allowed to conflict. Much as in work on memory models (e.g., [4]), these disciplines can be seen as contracts between the language implementation and the programmer: if a program conforms to certain restrictions, then the language implementation must run it with strong semantics. Such contracts should be "programmer-centric"—formulated in terms of programs and their high-level semantics, not of implementation details. The selection of particular restrictions represents a tradeoff.

– Stronger restrictions give more flexibility to the implementation by requiring it to run fewer programs with strong semantics An example of such a restriction is the imposition of a static type system that strictly segregates transacted and non-transacted memory (e.g., [9,2,12]). This segregation often implies the need to copy data across these two parts of memory.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 6–20, 2008.

– Conversely, weaker restrictions give more flexibility to the programmer but may enable fewer implementation strategies. For example, violation-freedom prohibits only programs whose executions cause conflicts at run-time, according to a high-level, strong, small-step operational semantics [2] (see also [8,3,5]). Violation-freedom does not consider lower-level conflicts that may arise in implementations with optimistic concurrency; so these implementations may not run all violation-free programs with strong semantics, and may therefore be disallowed.

We are exploring a new programming discipline that we call dynamic separation. Its basic idea is to distinguish memory locations that should be accessed transactionally from those that should be accessed directly, allowing this distinction to evolve dynamically in the course of program execution. The programmer (perhaps with the assistance of tools) indicates transitions between these modes. Dynamic separation restricts only where data is actually accessed by a program, not how the data is reachable through references.

Dynamic separation is intermediate between violation-freedom and static separation. Like violation-freedom, it does not require copying between two memory regions; like static separation, on the other hand, it enables implementations with weak atomicity, optimistic concurrency, lazy conflict detection, and in-place updates. Indeed, dynamic separation is compatible with a range of transactional-memory implementations. Moreover, dynamic separation does not necessitate changes in how non-transactional code is compiled. This property makes transactions "pay-to-use" and lets non-transacted code rely on features not available for re-compilation (cf., e.g., [15]).

A companion paper [1] studies dynamic separation informally. That paper provides a more detailed design rationale, an instantiation for C#, and some conceptually easy but useful refinements, in particular for read-only data. It also discusses implementations, describing our working implementation (done in the context of Bartok-STM [10]) and a variant that serves as a debugging tool for testing whether a program obeys the dynamic-separation discipline. As a case study, it examines the use of dynamic separation in the context of a concurrent web-proxy application built over an asynchronous IO library.

The present paper focuses on the formal definition and study of dynamic separation. It provides a precise formulation of dynamic separation, in the setting of a small calculus with transactions (Sections 2–5). It also establishes precise comparisons with static separation and with violation-freedom (Section 5). Furthermore, it considers two possible lower-level implementations of the calculus (Sections 6 and 7). One of the implementations relies on two heaps, with marshaling between them. The other includes optimistic concurrency and some other challenging features; it models important aspects of our Bartok-STM implementation. We establish the correctness of both implementations: we prove that, if a program conforms to the dynamic-separation discipline, then the two implementations will run it with strong semantics.

We present our results focusing on the Automatic Mutual Exclusion (AME) model [11,2] (Section 2). However, as explained in our companion paper, our

approach applies also to other models for programming with transactions, for instance to TIC [16].

2 AME and the AME Calculus

In this section we describe the AME programming model and the AME calculus, a small language with AME constructs that serves as the setting of our formal study. This section is mostly an informal review; in addition it introduces the new constructs for indicating transitions between modes, named `protect` and `unprotect`, into the AME calculus. We postpone a formal semantics of the calculus to Section 4.

2.1 AME

AME distinguishes "protected" code, which executes within transactions, from ordinary "unprotected" code. Importantly, the default is protected code.

Running an AME program consists in executing a set of asynchronous method calls. The AME system guarantees that the program execution is equivalent to executing each of these calls (or their atomic fragments, defined below) in some serialized order. The invocation `async MethodName(<method arguments>)` creates an asynchronous call. The caller continues immediately after this invocation. In the conceptual serialization of the program, the asynchronous callee will be executed after the caller has completed. AME achieves concurrency by executing asynchronous calls in transactions, overlapping the execution of multiple calls, with roll-backs when conflicts occur. If a transaction initiates other asynchronous calls, their execution is deferred until the initiating transaction commits, and they are discarded if the initiating transaction aborts.

Methods may contain invocations of `yield()`, which break an asynchronous call into multiple atomic fragments, implemented by committing one transaction and starting a new one. With this addition, the overall execution of a program is guaranteed to be a serialization of its atomic fragments.

Methods may also contain statements of the form `blockUntil(<p>)`, where `p` is a predicate. ¿From the programmer's perspective, an atomic fragment executes to completion only if all the predicates thus encountered in its execution evaluate to true. The implementation of `blockUntil(<p>)` does nothing if `p` holds; otherwise it aborts the current atomic fragment and retries it later.

In order to allow the use of legacy non-transacted code, AME provides block-structured `unprotected` sections. These must use existing mechanisms for synchronization. AME terminates the current atomic fragment before the code, and starts a new one afterwards.

2.2 The AME Calculus (with `protect` and `unprotect`)

The AME calculus is a small but expressive language that includes constructs for AME, higher-order functions, and imperative features. The following grammar

defines the syntax of the calculus, with the extensions required for dynamic separation.

$$
\begin{array}{rl}
V \in \mathit{Value} & = c \mid x \mid \lambda x.\, e \\
c \in \mathit{Const} & = \texttt{unit} \mid \texttt{false} \mid \texttt{true} \\
x, y \in \mathit{Var} & \\
e, f \in \mathit{Exp} & = V \mid e\ f \\
& \mid \texttt{ref}\ e \mid\ !e \mid e := f \\
& \mid \texttt{async}\ e \mid \texttt{blockUntil}\ e \\
& \mid \texttt{unprotected}\ e \\
& \mid \texttt{protect}\ e \mid \texttt{unprotect}\ e
\end{array}
$$

This syntax introduces syntactic categories of values, constants, variables, and expressions. The values are constants, variables, and lambda abstractions ($\lambda x.\, e$). In addition to values and to expressions of the forms `async` e, `blockUntil` e, and `unprotected` e, expressions include notations for function application (ef), allocation (`ref` e, which allocates a new reference location and returns it after initializing it to the value of e), dereferencing ($!e$, which returns the contents in the reference location that is the value of e), and assignment ($e := f$, which sets the reference location that is the value of e to the value of f). Expressions also include the new forms `protect` e and `unprotect` e, which evaluate e to a reference location, then make its value usable in transactions and outside transactions, respectively. We treat `yield` as syntactic sugar for `unprotected unit`. We write `let` $x = e$ `in` e' for $(\lambda x.\, e')\ e$, and write $e; e'$ for `let` $x = e$ `in` e' when x does not occur free in e'.

We make a small technical restriction that does not affect the expressiveness of the calculus: in any expression of the form `async` e, any occurrences of `unprotected` are under a λ. Thus, with our syntactic sugar, we can write `async` (`unit`; `unprotected` e'), but not `async` (`unprotected` e'). More generally, we can write `async` (`unit`; e'), for any e'. This technical restriction roughly ensures that an unprotected computation is not the first thing that happens in an asynchronous computation. It is needed only for Theorem 2, below.

3 An Example

This section presents an example, informally. Although this example is small and artificial, it serves to explain several aspects of our work. The example concerns the following code fragment:

```
let x = ref false in
let y = ref false in
let z = ref false in
async (x := true);
async (x := false; (blockUntil (!x)); y := true);
unprotected ((blockUntil (!y)); z := true)
```

This code first creates three reference locations, initialized to `false`, and binds x, y, and z to them, respectively. Then it forks two asynchronous executions. In

one, it sets x to `true`. In the other, it sets x to `false`, checks that x holds `true`, then sets y to `true`. In addition, the code contains an unprotected section that checks that y holds `true`, then sets z to `true`.

In reasoning about such code, programmers (and tools) should be entitled to rely on the high-level semantics of the AME constructs, without considering their possible implementation details. According to this high-level semantics, the two asynchronous executions are serialized. Therefore, the predicate $!x$ in the second asynchronous execution can never hold, so $y :=$ `true` is unreachable. Hence the predicate $!y$ in the unprotected section can never hold either, so z will never be set to `true`. The formal semantics of Section 4 justifies this reasoning.

On the other hand, lower-level implementations, such as that modeled in Section 7, may exhibit different, surprising behavior. With optimistic concurrency, the two asynchronous executions may be attempted simultaneously. For efficiency, updates to reference locations may be done in place, not buffered. So, if the assignment $x :=$ `true` immediately follows the assignment $x :=$ `false`, then the predicate $!x$ in the second asynchronous execution will hold, and $y :=$ `true` will execute. After the assignment $x :=$ `true`, the execution of (`blockUntil` $(!x)$); $y :=$ `true` is a "zombie" [7], doomed to roll back. With lazy conflict detection, a conflict may not yet be apparent. With weak atomicity, moreover, the unprotected section has an opportunity to execute, and the predicate $!y$ holds, so z will be set to `true`. When the two asynchronous executions attempt to commit, conflict detection will cause a roll-back of their effects on x and y, but not of the indirect effect on z. Therefore, the code may terminate with z holding `true`.

Despite the surprising behavior, we may want to allow such lower-level implementations because of their potential efficiency and compatibility with legacy code. So we may want to find criteria to exclude problematic programs. As indicated in the introduction, static separation is such a criterion; it statically segregates transacted and non-transacted memory. The code in our example does not obey static separation because (without dead-code elimination) y seems to be accessed both in a transaction and in the unprotected section. Unfortunately, static separation also forbids many reasonable code fragments, implying the need to marshal data back and forth between the two parts of memory.

Another possible criterion is violation-freedom. However, the code in our example is violation-free. In particular, according to the high-level semantics, there are no conflicting accesses to y at run-time, since $y :=$ `true` should never execute. Therefore, violation-freedom does not seem to be quite stringent enough to enable the use of some attractive implementation strategies.

Nevertheless, violation-free programs can often be instrumented with calls to `protect` and `unprotect` in order to conform to the dynamic-separation discipline. In this example, our particular formulation of dynamic separation requires adding two calls to `unprotect` in the last line of the code:

`unprotected` (`unprotect` y; `unprotect` z; (`blockUntil` $(!y)$); $z :=$ `true`)

Assuming that x, y, and z are initially in the mode where they are usable in transactions, we can reason that the placement of `unprotect` implies that x, y,

and z are always used in the appropriate mode, so the code does conform to the dynamic-separation discipline. In this reasoning, we need to consider only the behavior of the code in the high-level semantics. Although the high-level semantics of `unprotect` is quite straightforward—and resembles that of no-op—an implementation of `unprotect` may do non-trivial work. Sections 6 and 7 provide two illustrations of this point, in the latter case modeling important aspects of our actual implementation in Bartok-STM. In particular, `unprotect` y may block while y is being written in a transaction, even if the transaction is a zombie. Moreover, updating y in a transaction may check that y is protected. Crucially, neither of these implementation refinements require any changes to non-transactional access to y. In combination, these refinements can prevent the problematic behavior of this code, guaranteeing that it runs correctly.

Zombies constitute only one of several problems in this area. Others include the so-called privatization and publication problems (e.g., [2,17]). Although we do not discuss those in detail, our approach and our results address them as well. In particular, the correctness theorems below imply that publication and privatization idioms can execute correctly.

4 Semantics

The strong semantics of the AME calculus is a small-step operational semantics in which at most one transaction may take steps at any one time, and non-transacted code may take steps only when there is no current transaction taking steps [2]. We extend this strong semantics to the new constructs.

States. A state $\langle \sigma, \tau, T, e \rangle$ consists of a reference store σ, a protection state τ, a collection of expressions T, and a distinguished active expression e. A reference store σ is a finite mapping of reference locations to values. Similarly, a protection state τ is a finite mapping of reference locations to protection modes, which we write `P` and `U`. It is a "history variable", in the sense that it is determined by the history of execution and does not influence this history. Reference locations are simply special kinds of variables that can be bound only by the respective store and protection state. We write *RefLoc* for the set of reference locations; we assume that *RefLoc* is infinite. For every state $\langle \sigma, \tau, T, e \rangle$, we require that $dom(\sigma) = dom(\tau)$ and, if $r \in RefLoc$ occurs in $\langle \sigma, \tau, T, e \rangle$, then $r \in dom(\sigma)$. We set:

$$\begin{aligned} S \in \quad & State \subset RefStore \times ProtState \times ExpSeq \times Exp \\ \sigma \in \quad & RefStore = RefLoc \rightharpoonup Value \\ \tau \in \quad & ProtState = RefLoc \rightharpoonup \{\mathtt{P}, \mathtt{U}\} \\ r \in \quad & RefLoc \subset Var \\ T \in \quad & ExpSeq = Exp^* \end{aligned}$$

Steps. 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 an expression e,

$$\langle \sigma, \tau, T, \mathcal{P}[\ (\lambda x.\ e)\ V\ ] \rangle \longmapsto_s \langle \sigma, \tau, T, \mathcal{P}[\ e[V/x]\ ] \rangle \quad \text{(Trans Appl P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{U}[\ e[V/x]\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Appl U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ \mathtt{ref}\ V\ ] \rangle \longmapsto_s \langle \sigma[r \mapsto V], \tau[r \mapsto \mathtt{P}], T, \mathcal{P}[\ r\ ] \rangle \quad \text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma) \quad \text{(Trans Ref P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \mathtt{ref}\ V\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma[r \mapsto V], \tau[r \mapsto \mathtt{U}], T.\mathcal{U}[\ r\ ].T', \mathtt{unit} \rangle \quad \text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma) \quad \text{(Trans Ref U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ !r\ ] \rangle \longmapsto_s \langle \sigma, \tau, T, \mathcal{P}[\ V\ ] \rangle \quad \text{if } \sigma(r) = V \quad \text{(Trans Deref P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ !r\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{U}[\ V\ ].T', \mathtt{unit} \rangle \quad \text{if } \sigma(r) = V \quad \text{(Trans Deref U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ r := V\ ] \rangle \longmapsto_s \langle \sigma[r \mapsto V], \tau, T, \mathcal{P}[\ \mathtt{unit}\ ] \rangle \quad \text{(Trans Set P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ r := V\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma[r \mapsto V], \tau, T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Set U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ \mathtt{async}\ e\ ] \rangle \longmapsto_s \langle \sigma, \tau, e.T, \mathcal{P}[\ \mathtt{unit}\ ] \rangle \quad \text{(Trans Async P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \mathtt{async}\ e\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma, \tau, e.T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Async U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ \mathtt{blockUntil\ true}\ ] \rangle \longmapsto_s \langle \sigma, \tau, T, \mathcal{P}[\ \mathtt{unit}\ ] \rangle \quad \text{(Trans Block P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \mathtt{blockUntil\ true}\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Block U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ \mathtt{unprotected}\ e\ ] \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{P}[\ \mathtt{unprotected}\ e\ ], \mathtt{unit} \rangle \quad \text{(Trans Unprotect)}_s$$

$$\langle \sigma, \tau, T.\mathcal{E}[\ \mathtt{unprotected}\ V\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{E}[\ V\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Close)}_s$$

$$\langle \sigma, \tau, T.e.T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.T', e \rangle \quad \text{(Trans Activate)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \mathtt{protect}\ r\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma, \tau[r \mapsto \mathtt{P}], T.\mathcal{U}[\ r\ ].T', \mathtt{unit} \rangle \quad \text{(Trans DynP)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \mathtt{unprotect}\ r\ ].T', \mathtt{unit} \rangle \longmapsto_s \langle \sigma, \tau[r \mapsto \mathtt{U}], T.\mathcal{U}[\ r\ ].T', \mathtt{unit} \rangle \quad \text{(Trans DynU)}_s$$

Fig. 1. Transition rules with dynamic separation

we write $\mathcal{C}[\ e\ ]$ for the result of placing e in the hole in $\mathcal{C}$. We use several kinds of evaluation contexts, defined by:

$$\begin{aligned}
\mathcal{P} &= [\ ] \mid \mathcal{P}\ e \mid V\ \mathcal{P} \mid \mathtt{ref}\ \mathcal{P} \mid !\mathcal{P} \mid \mathcal{P} := e \mid r := \mathcal{P} \mid \mathtt{blockUntil}\ \mathcal{P} \\
&\quad \mid \mathtt{protect}\ \mathcal{P} \mid \mathtt{unprotect}\ \mathcal{P} \\
\mathcal{U} &= \mathtt{unprotected}\ \mathcal{E} \mid \mathcal{U}\ e \mid V\ \mathcal{U} \mid \mathtt{ref}\ \mathcal{U} \mid !\mathcal{U} \mid \mathcal{U} := e \mid r := \mathcal{U} \mid \mathtt{blockUntil}\ \mathcal{U} \\
&\quad \mid \mathtt{protect}\ \mathcal{U} \mid \mathtt{unprotect}\ \mathcal{U} \\
\mathcal{E} &= [\ ] \mid \mathcal{E}\ e \mid V\ \mathcal{E} \mid \mathtt{ref}\ \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \mid \mathtt{blockUntil}\ \mathcal{E} \\
&\quad \mid \mathtt{unprotected}\ \mathcal{E} \mid \mathtt{protect}\ \mathcal{E} \mid \mathtt{unprotect}\ \mathcal{E}
\end{aligned}$$

A context $\mathcal{E}$ is a general evaluation context; a context $\mathcal{U}$ is one where the hole is under `unprotected`; a context $\mathcal{P}$ is one where it is not.

Figure 1 gives rules that specify the transition relation that takes execution from one state to the next. In these rules, we write $e[V/x]$ for the result of the capture-free substitution of V for x in e, and write $\sigma[r \mapsto V]$ for the store that agrees with σ except at r, which is mapped to V. The subscript s in $\longmapsto_s$ indicates that this is a strong semantics.

In rules (Trans Ref P)$_s$ and (Trans Ref U)$_s$, the reference-allocation construct `ref` e initializes the new location's mode to `P` (when allocating inside a transaction) or to `U` (otherwise). In rules (Trans DynP)$_s$ and (Trans DynU)$_s$, the new constructs `protect` and `unprotect` set the mode to `P` and to `U` respectively. It is not an error to call `protect` on a reference location already in mode `P`. Similarly,

it is not an error to call `unprotect` on a reference location already in mode `U`. This design choice enables a broader range of implementations, as discussed in our companion paper.

According to the rules, `protect` and `unprotect` work only outside transactions. They get stuck otherwise. Fundamentally, we do not want to rely on `protect` and `unprotect` in transactions because of questionable interactions, such as the possibility of zombie updates to the protection state.

5 The Dynamic-Separation Discipline

We give a precise definition of dynamic separation. We also establish results that relate dynamic separation to static separation and to violation-freedom.

5.1 Definition

The definition of dynamic separation says that, in the course of an execution, reads and writes to a reference location should happen only if the protection state of the reference location is consistent with the context of the operation. The definition is intended to constrain expressions, but more generally it applies to initial states of executions.

Given a state $\langle \sigma, \tau, T, e \rangle$, a read or a write may occur in two cases:

- e is of the form $\mathcal{P}[\ !r\ ]$ or $\mathcal{P}[\ r := V\ ]$; or
- $e = \texttt{unit}$ and T contains an expression of the form $\mathcal{U}[\ !r\ ]$ or $\mathcal{U}[\ r := V\ ]$.

Accordingly, we say that a state S obeys the dynamic-separation discipline, and write $\mathcal{DS}(S)$, if whenever $S \longmapsto_s^* \langle \sigma, \tau, T, e \rangle$, the state $\langle \sigma, \tau, T, e \rangle$ is such that:

- if e is of the form $\mathcal{P}[\ !r\ ]$ or $\mathcal{P}[\ r := V\ ]$, then $\tau(r) = \texttt{P}$;
- if $e = \texttt{unit}$ and T contains an expression of the form $\mathcal{U}[\ !r\ ]$ or $\mathcal{U}[\ r := V\ ]$, then $\tau(r) = \texttt{U}$.

In sum, a state S obeys the dynamic-separation discipline if, in S, reads or writes to a reference location r can happen only if r's protection state (`P` or `U`) is consistent with the context (transacted or not, respectively) of the operation, and if the same is true for any state reachable from S.

5.2 Comparison with Static Separation

Static separation can be defined as a type system; its details are straightforward, and for AME they are given in [2, Section 6.2]. There, the judgment $E \vdash \langle \sigma, T, e \rangle$ says that the state $\langle \sigma, T, e \rangle$ obeys the static-separation discipline in a typing environment E, which gives types of the form $\texttt{Ref}_\texttt{P}\ t$ or $\texttt{Ref}_\texttt{U}\ t$ for the free reference locations of the state. The state does not include a protection state τ, since separation is static. Given E, however, we write τ_E for the protection state that maps each reference location to `P` or `U` according to its type in E. We obtain:

Theorem 1. *If* $E \vdash \langle \sigma, T, e \rangle$ *then* $\mathcal{DS}(\langle \sigma, \tau_E, T, e \rangle)$.

The converse of this theorem is false, not only because of possible occurrences of `protect` and `unprotect` but also because of examples like that of Section 3.

5.3 Comparison with Violation-Freedom

As discussed above, violation-freedom is a condition that prohibits programs whose executions cause certain conflicts at run-time. More precisely, we say that a state $\langle \sigma, \tau, T, e \rangle$ has a violation on r when:

- e is of the form $\mathcal{P}[\ e'\ ]$,
- T contains an expression of the form $\mathcal{U}[\ e''\ ]$,
- e' and e'' are of the form $!r$ or $r := V$ for some V, and at least one is of the latter form.

(Note that the second of these clauses does not require $e = \texttt{unit}$, unlike the corresponding part of the definition of dynamic separation.) We say that a state S obeys the violation-freedom discipline, and write $\mathcal{VF}(S)$, if whenever $S \longmapsto_s^* S'$, the state S' does not have violations on any r.

In general, dynamic separation is not sufficient for violation-freedom. For instance, the state

$$\langle \emptyset[r \mapsto \texttt{false}], \emptyset[r \mapsto \texttt{P}], \texttt{unprotected}\ (r := \texttt{true}), \texttt{blockUntil}\ !r \rangle$$

obeys the dynamic-separation discipline, but has an obvious violation on r. This violation never leads to an actual concurrent access under the strong semantics.

Dynamic separation does however imply violation-freedom for initial states of the form $\langle \sigma, \tau, T, \texttt{unit} \rangle$, in which there is no active transaction—but of course a transaction may be activated. We regard this result, together with Theorem 1, as proof of our informal statement that dynamic separation is intermediate between violation-freedom and static separation.

Theorem 2. *If* $\mathcal{DS}(\langle \sigma, \tau, T, \texttt{unit} \rangle)$*, then* $\mathcal{VF}(\langle \sigma, \tau, T, \texttt{unit} \rangle)$.

Conversely, violation-freedom is not a sufficient condition for dynamic separation, for several reasons. Most obviously, violation-freedom does not require the use of explicit calls to `protect` and `unprotect`. In addition, violation-freedom does not constrain read-read concurrency, while dynamic separation does. Strengthening violation-freedom so that it also constrains read-read concurrency, we have developed a method for taking a violation-free expression and adding calls to `protect` and `unprotect` so as to make it obey dynamic separation. We omit the details of our method, but briefly note its two main assumptions: (1) The method requires the absence of race conditions in unprotected computations, because race conditions could cause instrumentation to work incorrectly. (2) It also assumes that we can distinguish transacted and non-transacted code at instrumentation time; code duplication can make this task trivial.

6 An Implementation with Two Heaps

In this section, we consider an abstract machine with two separate heaps accessed by transactional and non-transactional code, respectively. The constructs

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ (\lambda x.\ e)\ V\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ e[V/x]\ ] \rangle \quad \text{(Trans Appl P)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ e[V/x]\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Appl U)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \mathtt{ref}\ V\ ] \rangle \longmapsto_t \langle \sigma_1[r \mapsto V], \sigma_2[r \mapsto V], \tau[r \mapsto \mathtt{P}], T, \mathcal{P}[\ r\ ] \rangle \quad \text{(Trans Ref P)}_t$$
$$\text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma_1)$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \mathtt{ref}\ V\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1[r \mapsto V], \sigma_2[r \mapsto V], \tau[r \mapsto \mathtt{U}], T.\mathcal{U}[\ r\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Ref U)}_t$$
$$\text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma_1)$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ !r\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ V\ ] \rangle \quad \text{(Trans Deref P)}_t$$
$$\text{if } \sigma_1(r) = V$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ !r\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ V\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Deref U)}_t$$
$$\text{if } \sigma_2(r) = V$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ r := V\ ] \rangle \longmapsto_t \langle \sigma_1[r \mapsto V], \sigma_2, \tau, T, \mathcal{P}[\ \mathtt{unit}\ ] \rangle \quad \text{(Trans Set P)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ r := V\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2[r \mapsto V], \tau, T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Set U)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \mathtt{async}\ e\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, e.T, \mathcal{P}[\ \mathtt{unit}\ ] \rangle \quad \text{(Trans Async P)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \mathtt{async}\ e\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, e.T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Async U)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \mathtt{blockUntil\ true}\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \mathtt{unit}\ ] \rangle \quad \text{(Trans Block P)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \mathtt{blockUntil\ true}\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Block U)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \mathtt{unprotected}\ e\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{P}[\ \mathtt{unprotected}\ e\ ], \mathtt{unit} \rangle \quad \text{(Trans Unprotect)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{E}[\ \mathtt{unprotected}\ V\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{E}[\ V\ ].T', \mathtt{unit} \rangle \quad \text{(Trans Close)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.e.T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.T', e \rangle \quad \text{(Trans Activate)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \mathtt{protect}\ r\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ r\ ].T', \mathtt{unit} \rangle \quad \text{(Trans DynP (1))}_t$$
$$\text{if } \tau(r) = \mathtt{P}$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \mathtt{protect}\ r\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1[r \mapsto \sigma_2(r)], \sigma_2, \tau[r \mapsto \mathtt{P}], T.\mathcal{U}[\ r\ ].T', \mathtt{unit} \rangle \quad \text{(Trans DynP (2))}_t$$
$$\text{if } \tau(r) = \mathtt{U}$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \mathtt{unprotect}\ r\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2[r \mapsto \sigma_1(r)], \tau[r \mapsto \mathtt{U}], T.\mathcal{U}[\ r\ ].T', \mathtt{unit} \rangle \quad \text{(Trans DynU (1))}_t$$
$$\text{if } \tau(r) = \mathtt{P}$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \mathtt{unprotect}\ r\ ].T', \mathtt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ r\ ].T', \mathtt{unit} \rangle \quad \text{(Trans DynU (2))}_t$$
$$\text{if } \tau(r) = \mathtt{U}$$

Fig. 2. Transition rules with two heaps

`protect` and `unprotect` marshal between these heaps. Although this two-heap scheme is not particularly efficient, it is reminiscent of some practical systems that use different data formats in transactional and non-transactional code. It is also an interesting approximation of a static-separation regime, and illustrates that `protect` and `unprotect` may do more than in the high-level semantics of Figure 1. Still, for expressions that obey the dynamic-separation discipline, we prove that this two-heap implementation respects the high-level semantics.

6.1 Operational Semantics

We define the two-heap implementation as a lower-level semantics, in the style of that of Section 4 though with some additional intricacies.

States. The components of a state are much like those in Section 4, except that there are two reference stores rather than one. A state $\langle \sigma_1, \sigma_2, \tau, T, e \rangle$ consists of two reference stores σ_1 and σ_2, a protection state τ, a collection of expressions T,

and a distinguished active expression e. We require that $dom(\sigma_1) = dom(\sigma_2) = dom(\tau)$ and that, if $r \in \mathit{RefLoc}$ occurs in the state, then $r \in dom(\sigma_1)$. So we set:

$$S \in \mathit{State} \subset \mathit{RefStore} \times \mathit{RefStore} \times \mathit{ProtState} \times \mathit{ExpSeq} \times \mathit{Exp}$$

Steps. Figure 2 gives rules that specify the transition relation of this semantics. According to these rules, `ref` e sets the protection state of a new reference location r and initializes the contents of r in each of the reference stores. Initializing the contents in the appropriate reference store would suffice, provided r is added to the domain of both reference stores. While reading or writing a location, the context in which an expression executes determines which reference store it accesses. Finally, `protect` r and `unprotect` r perform marshaling, as follows. If r already has the desired protection state, then no copying is required. (In fact, copying could overwrite fresh contents with stale ones.) Otherwise, r's contents are copied from one reference store to the other.

6.2 Correctness

The two-heap implementation is correct under the dynamic-separation discipline, in the following sense:

Theorem 3. *Assume that* $\mathcal{DS}(\langle \sigma, \tau, T, e \rangle)$*, that* $dom(\sigma) = dom(\sigma_1) = dom(\sigma_2)$*, and that* $\sigma_1(r) = \sigma(r)$ *if* $\tau(r) = \mathtt{P}$ *and* $\sigma_2(r) = \sigma(r)$ *if* $\tau(r) = \mathtt{U}$*. Consider a computation with two heaps:*

$$\langle \sigma_1, \sigma_2, \tau, T, e \rangle \longmapsto_t^* \langle \sigma_1', \sigma_2', \tau', T', e' \rangle$$

Then there is a computation:

$$\langle \sigma, \tau, T, e \rangle \longmapsto_s^* \langle \sigma', \tau', T', e' \rangle$$

for some σ' *such that* $dom(\sigma') = dom(\sigma_1') = dom(\sigma_2')$ *and, for every* $r \in dom(\sigma')$*, if* $\tau'(r) = \mathtt{P}$*, then* $\sigma_1'(r) = \sigma'(r)$*, and if* $\tau'(r) = \mathtt{U}$*, then* $\sigma_2'(r) = \sigma'(r)$*.*

This simulation result implies that the contents of a reference location r is always correct in the reference store that corresponds to r's current protection state. The dynamic-separation hypothesis is essential: it is required for extending the simulation in the cases of (Trans Deref $\ldots)_t$ and (Trans Set $\ldots)_t$. Without it, the execution with two heaps may produce incorrect results.

7 An Implementation with Optimistic Concurrency

Going further, we treat a lower-level implementation in which multiple transactions execute simultaneously, with roll-backs in case of conflict. This implementation is based on one studied in our previous work [2], with the addition of dynamic separation. As explained there, various refinements are possible, but they are not necessary for our present purposes. Our goal is to show how dynamic separation works (correctly) in a setting with realistic, challenging features such as in-place updates (e.g., [13]). The model developed in this section is an abstract version of our actual implementation in Bartok-STM.

7.1 Operational Semantics

Again, we define the implementation as a lower-level semantics.

States. States become more complex for this semantics. In addition to the components σ, τ, and T that appear in the earlier semantics, we add constructs for roll-back and optimistic concurrency. In order to support, roll-back, we maintain a log l of the reference locations that have been modified, with their corresponding original values. In the case of roll-back, we use the log to restore these values in the reference store. For optimistic concurrency, we have a list of tuples instead of a single active expression. Each of the tuples is called a try, and consists of the following components:

- an active expression e,
- another expression f from which e was obtained (its "origin"),
- a description of the accesses that e has performed, which are used for conflict detection and which here is simply a list of reference locations,
- a list P of threads to be forked upon commit.

For every state $\langle \sigma, \tau, T, O, l \rangle$, we require that $dom(\sigma) = dom(\tau)$ and that, if $r \in \mathit{RefLoc}$ occurs in the state, then $r \in dom(\sigma)$. We set:

$$\begin{aligned}
S \in \quad & \mathit{State} \subset \mathit{RefStore} \times \mathit{ProtState} \times \mathit{ExpSeq} \times \mathit{TrySeq} \times \mathit{Log} \\
\sigma \in \quad & \mathit{RefStore} = \mathit{RefLoc} \rightharpoonup \mathit{Value} \\
\tau \in \quad & \mathit{ProtState} = \mathit{RefLoc} \rightharpoonup \{\mathtt{P}, \mathtt{U}\} \\
l \in \quad & \mathit{Log} = (\mathit{RefLoc} \times \mathit{Value})^* \\
r \in \quad & \mathit{RefLoc} \subset \mathit{Var} \\
T, P \in \quad & \mathit{ExpSeq} = \mathit{Exp}^* \\
O \in \quad & \mathit{TrySeq} = \mathit{Try}^* \\
d \in \quad & \mathit{Try} = \mathit{Exp} \times \mathit{Exp} \times \mathit{Accesses} \times \mathit{ExpSeq} \\
a \in \quad & \mathit{Accesses} = \mathit{RefLoc}^*
\end{aligned}$$

Steps. Figure 3 gives the rules of this semantics, relying on these definitions:

- (e_i, f_i, a_i, P_i) and (e_j, f_j, a_j, P_j) conflict if a_i and a_j have at least one element in common.
- (e, f, a, P) conflicts with O if (e, f, a, P) conflicts with some try in O.
- Given a log l and a list of reference locations a, $l - a$ is the log obtained from l by restricting to reference locations not in a.
- If O is $(e_1, f_1, a_1, P_1).\cdots.(e_n, f_n, a_n, P_n)$ then $\mathit{origin}(O)$ is the list $f_1.\cdots.f_n$.
- σl is the result of applying all elements of l to σ.

Many aspects of this semantics are explained in our previous work. Here we focus on the new ones, namely those related to dynamic separation.

Rule (Trans DynU)$_o$ requires that, when a reference location is unprotected, it is not being written by any try. This restriction is a formalization of one present in our Bartok-STM implementation (where "being written" means, more specifically, "open for update"). The restriction on (Trans DynU)$_o$ can be satisfied

$\langle \sigma, \tau, T, O.(\mathcal{P}[\ (\lambda x.\ e)\ V\ ], f, a, P).O', l\rangle \longmapsto_o \langle \sigma, \tau, T, O.(\mathcal{P}[\ e[V/x]\ ], f, a, P).O', l\rangle$ (Trans Appl P)$_o$

$\langle \sigma, \tau, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau, T.\mathcal{U}[\ e[V/x]\ ].T', O, l\rangle$ (Trans Appl U)$_o$

$\langle \sigma, \tau, T, O.(\mathcal{P}[\ \texttt{ref}\ V\ ], f, a, P).O', l\rangle \longmapsto_o \langle \sigma[r \mapsto V], \tau[r \mapsto \texttt{P}], T, O.(\mathcal{P}[\ r\ ], f, a, P).O', l\rangle$ (Trans Ref P)$_o$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$

$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', O, l\rangle \longmapsto_o \langle \sigma[r \mapsto V], \tau[r \mapsto \texttt{U}], T.\mathcal{U}[\ r\ ].T', O, l\rangle$ (Trans Ref U)$_o$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$

$\langle \sigma, \tau, T, O.(\mathcal{P}[\ !r\ ], f, a, P).O', l\rangle \longmapsto_o \langle \sigma, \tau, T, O.(\mathcal{P}[\ V\ ], f, r.a, P).O', l\rangle$ (Trans Deref P)$_o$
if $\sigma(r) = V$

$\langle \sigma, \tau, T.\mathcal{U}[\ !r\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau, T.\mathcal{U}[\ V\ ].T', O, l\rangle$ (Trans Deref U)$_o$
if $\sigma(r) = V$

$\langle \sigma, \tau, T, O.(\mathcal{P}[\ r := V\ ], f, a, P).O', l\rangle \longmapsto_o \langle \sigma[r \mapsto V], \tau, T, O.(\mathcal{P}[\ \texttt{unit}\ ], f, r.a, P).O', l'\rangle$ (Trans Set P)$_o$
where $l' = \text{if } r \in \mathit{dom}(l) \text{ then } l \text{ else } l.[r \mapsto \sigma(r)]$
and $\tau(r) = \texttt{P}$

$\langle \sigma, \tau, T.\mathcal{U}[\ r := V\ ].T', O, l\rangle \longmapsto_o \langle \sigma[r \mapsto V], \tau, T.\mathcal{U}[\ \texttt{unit}\ ].T', O, l\rangle$ (Trans Set U)$_o$

$\langle \sigma, \tau, T, O.(\mathcal{P}[\ \texttt{async}\ e\ ], f, a, P).O', l\rangle \longmapsto_o \langle \sigma, \tau, T, O.(\mathcal{P}[\ \texttt{unit}\ ], f, a, e.P).O', l\rangle$ (Trans Async P)$_o$

$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{async}\ e\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau, e.T.\mathcal{U}[\ \texttt{unit}\ ].T', O, l\rangle$ (Trans Async U)$_o$

$\langle \sigma, \tau, T, O.(\mathcal{P}[\ \texttt{blockUntil true}\ ], f, a, P).O', l\rangle \longmapsto_o \langle \sigma, \tau, T, O.(\mathcal{P}[\ \texttt{unit}\ ], f, a, P).O', l\rangle$ (Trans Block P)$_o$

$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{blockUntil true}\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau, T.\mathcal{U}[\ \texttt{unit}\ ].T', O, l\rangle$ (Trans Block U)$_o$

$\langle \sigma, \tau, T, O, l\rangle \longmapsto_o \langle \sigma l, \tau, \mathit{origin}(O).T, \emptyset, \emptyset\rangle$ (Trans Undo)$_o$

$\langle \sigma, \tau, T, O.(\mathcal{P}[\ \texttt{unprotected}\ e\ ], f, a, P).O', l\rangle \longmapsto_o \langle \sigma, \tau, T.\mathcal{P}[\ \texttt{unprotected}\ e\ ].P, O.O', l - a\rangle$ (Trans Unprotect)$_o$
if $(\mathcal{P}[\ \texttt{unprotected}\ e\ ], f, a, P)$ does not conflict with $O.O'$

$\langle \sigma, \tau, T, O.(\texttt{unit}, f, a, P).O', l\rangle \longmapsto_o \langle \sigma, \tau, T.P, O.O', l - a\rangle$ (Trans Done)$_o$
if $(\texttt{unit}, f, a, P)$ does not conflict with $O.O'$

$\langle \sigma, \tau, T.\mathcal{E}[\ \texttt{unprotected}\ V\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau, T.\mathcal{E}[\ V\ ].T', O, l\rangle$ (Trans Close)$_o$

$\langle \sigma, \tau, T.e.T', O, l\rangle \longmapsto_o \langle \sigma, \tau, T.T', (e, e, \emptyset, \emptyset).O, l\rangle$ (Trans Activate)$_o$

$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{protect}\ r\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau[r \mapsto \texttt{P}], T.\mathcal{U}[\ r\ ].T', O, l\rangle$ (Trans DynP)$_o$

$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{unprotect}\ r\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau[r \mapsto \texttt{U}], T.\mathcal{U}[\ r\ ].T', O, l\rangle$ (Trans DynU)$_o$
if $r \notin \mathit{dom}(l)$

Fig. 3. Transition rules with optimistic concurrency and dynamic separation

by performing an undo. However, an undo is never forced to happen. Indeed, the rules allow undo to happen at any point—possibly but not necessarily when there is a conflict. Conflict detection may be eager or lazy; the rules do not impose a particular strategy in this respect.

There is no corresponding subtlety in rule (Trans DynP)$_o$. Bartok-STM employs a more elaborate version of this rule in order to allow compiler optimizations that reorder accesses.

When writing to a reference location from within a transaction (rule (Trans Set P)$_o$), the protection state of that reference location is verified. Even with dynamic separation, this check is essential for correctness because of the possibility of zombie transactions. On the other hand, a check is not needed for reads (rule (Trans Deref P)$_o$), nor for accesses in unprotected code (rules (Trans Deref U)$_o$ and (Trans Set U)$_o$). These features of the rules correspond to important aspects of our Bartok-STM implementation, which aims to allow the re-use of legacy code without instrumentation.

7.2 Correctness

The implementation with optimistic concurrency is correct with respect to the strong semantics of Section 4, in the following sense:

Theorem 4. *Assume that* $\mathcal{DS}(\langle \sigma, \tau, T, \mathtt{unit} \rangle)$. *Consider a computation:*

$$\langle \sigma, \tau, T, \emptyset, \emptyset \rangle \longmapsto_o^* \langle \sigma', \tau', T', \emptyset, \emptyset \rangle$$

Then there is a computation:

$$\langle \sigma, \tau, T, \mathtt{unit} \rangle \longmapsto_s^* \langle \sigma'', \tau'', T'', \mathtt{unit} \rangle$$

for some σ'', τ'', *and* T'' *such that* σ' *is an extension of* σ'', τ' *is an extension of* τ'', *and* $T'' = T'$ *up to reordering.*

Much as for Theorem 3, the dynamic-separation assumption is essential for Theorem 4. However, Theorem 4 is much harder than Theorem 3.

8 Conclusion

A notable aspect of our research on AME is that we have developed formal semantics alongside our software artifacts. The formal semantics have helped guide the practical implementation work and vice versa. As in the present study of dynamic separation, formal semantics shed light on the behavior of constructs and the properties of programming disciplines, even in the face of diverse implementation techniques.

Our objective is to enable the creation of programs by programmers with normal (not exceptional) skills, such that the programs will be satisfactory on current and future hardware, especially multi-processor and multi-core hardware. The programs must be semantically correct and must actually run correctly—at least the semantics and the implementations should be well-defined and simple enough that they are not an obstacle to correctness. The programs should also be efficient, so they should utilize concurrency where appropriate. Transactional memory with dynamic separation appears as a promising element in reconciling these goals.

Acknowledgements. This work was done at Microsoft Research. We are grateful to Katie Coons, Rebecca Isaacs, Yossi Levanoni, and JP Martin for helpful discussions and comments, and to Andrew Birrell, Johnson Hsieh, and Michael Isard for our joint work, which gave rise to the present 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., 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, pp. 63–74 (2008)

3. Adl-Tabatabai, A.-R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for efficient software transactional memory. In: PLDI 2006: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 26–37 (2006)
4. Adve, S.V.: Designing memory consistency models for shared-memory multiprocessors. PhD thesis, U. Wisconsin–Madison (1993)
5. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress language specification, v1.0β. Technical report, Sun Microsystems (March 2007)
6. Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing transactional semantics: The subtleties of atomicity. In: Proc. 2005 Workshop on Duplicating, Deconstructing and Debunking (2005)
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
8. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 388–402 (2003)
9. 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, pp. 48–60 (2005)
10. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: PLDI 2006: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 14–25 (2006)
11. Isard, M., Birrell, A.: Automatic mutual exclusion. In: Proc. 11th Workshop on Hot Topics in Operating Systems (May 2007)
12. 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, pp. 51–62 (2008)
13. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP 2006: Proc. Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 187–197 (2006)
14. Shavit, N., Touitou, D.: Software transactional memory. In: Proc. 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213 (August 1995)
15. Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing isolation and ordering in STM. In: PLDI 2007: Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 78–88 (2007)
16. Smaragdakis, Y., Kay, A., Behrends, R., Young, M.: Transactions with isolation and cooperation. In: Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 191–210 (2007)
17. Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. Technical Report 915, CS Dept, U. Rochester (2007)

Completeness and Nondeterminism in Model Checking Transactional Memories*

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh

EPFL, Switzerland

Abstract. Software transactional memory (STM) offers a disciplined concurrent programming model for exploiting the parallelism of modern processor architectures. This paper presents the first *deterministic* specification automata for strict serializability and opacity in STMs. Using an antichain-based tool, we show our deterministic specifications to be equivalent to more intuitive, nondeterministic specification automata (which are too large to be determinized automatically). Using deterministic specification automata, we obtain a *complete* verification tool for STMs. We also show how to model and verify contention management within STMs. We automatically check the opacity of popular STM algorithms, such as TL2 and DSTM, with a universal contention manager. The universal contention manager is *nondeterministic* and establishes correctness for all possible contention management schemes.

1 Introduction

Software transactional memory (STM) has gained much recent interest with the advent of multicore architectures. An STM enables the programmer to structure her application in terms of coarse-grained code blocks that appear to be executed atomically [7,12]. Behind the apparent simplicity of the STM abstraction, however, lie challenging algorithms that seek to ensure transactional atomicity without restricting parallelism. Despite the large amount of experimental work on such algorithms [8], little effort has been devoted to their formalization [3,11].

We believe that an approach to formalizing and verifying STM algorithms can only have impact if it is accepted by the transactional memory community, and this concern has guided our decisions in choosing the correctness properties that STMs should satisfy. For this reason we consider strict serializability [9] and opacity [3] as the two measures of the correctness of STMs. The former requires committed transactions to appear as if executed at indivisible points in time during their lifetime. Opacity goes a step further and also requires aborted transactions to always access consistent state. The notion of opacity corresponds closest to an emerging consensus about correctness in the transactional software community [2,6]. The motivation of this work is to formally check popular STM algorithms such as DSTM [6] and TL2 [2] against opacity.

* This research was supported by the Swiss National Science Foundation.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 21–35, 2008.

Our first step in this direction addressed the problem of space explosion in STMs [4]. We restricted our attention to STMs that satisfy certain structural properties, and we proved that the correctness of such an STM for 2 threads and 2 variables implies the correctness of the STM for an arbitrary number of threads and variables. Then, to check the correctness of an STM for 2 threads and 2 variables, we modeled an STM as a *deterministic* transition system. At the same time, we constructed *nondeterministic* specification automata for the strictly serializable and opaque words on 2 threads and 2 variables. An STM is then correct if the language of the STM transition system is included in the language of the specification automaton. Since checking language inclusion was too expensive, we resorted to checking the existence of a simulation relation. As the existence of a simulation relation is a sufficient, but not a necessary, condition for language inclusion with nondeterministic specifications, our procedure was sound but not complete.

In this paper, we provide *deterministic* specification automata for strict serializability and opacity. Constructing such deterministic specifications is non-trivial. Roughly speaking, the difficulty comes in specifying opacity in the presence of aborting transactions. In this scenario, some conflicts between transactions are transitive, whereas others are not. The determinism of the specification automata allows for an efficient check of language inclusion (by constructing the product of the specification and implementation), which results in a complete verification procedure. Moreover —and perhaps surprisingly— the deterministic specification automata are significantly smaller than their nondeterministic counterparts, which provide more intuitive specifications. As the nondeterministic automata are too large to be determinized explicitly, we use an antichain-based tool [13] to prove the correctness of our deterministic specifications. The tool shows language equivalence of our deterministic automata with the natural, nondeterministic specifications, without an explicit subset construction. The smaller, deterministic specification automata speed up the verification of STMs like DSTM and TL2 by an order of magnitude. This speed-up allows us to check the correctness of STMs with much larger state spaces. We use this gain to verify *nondeterministic* STMs that model realistic contention management schemes like exponential backoff and prioritized transactions.

In practice, STMs employ an external *contention manager* to enhance liveness [5,10]. The idea of the contention manager is to resolve conflicts between transactions on the basis of their past behavior. Various contention managers have been proposed in the literature. For example, the *Karma* contention manager prioritizes transactions according to the number of objects opened, whereas the *Polite* contention manager backs off conflicting transactions for a random duration [10]. For verification purposes, modeling a contention manager explicitly is infeasible. First, it would blow up the state space, as the decision of a contention manager often depends on the past behavior of every thread in an intricate manner. Second, many contention managers break the structural properties that the model checking approach [4] expects in order to reduce the problem to two threads and two variables. Third, an STM is designed to maintain

safety *for all* possible contention managers, which can be changed independent of the STM.

To tackle these issues, we model the effect of all possible contention managers on an STM by defining a *universal* contention manager. An STM with the universal contention manager is a nondeterministic transition system that contains transitions for all possible decisions of any contention manager. Moreover, the universal contention manager does not break any structural property of the STM, which allows us to reduce the verification problem to two threads and two variables. Putting everything together, we are able to automatically verify opacity for STMs such as DSTM and TL2 *for all* contention managers.

Related work. This work improves the model-checking approach [4] for transactional memories in terms of both the generality of the model (including nondeterministic contention management) and the efficiency and completeness of the verification procedure. There also has been recent independent work on the formal verification of STM algorithms [1]. That verification model checks STMs applied to programs with a small number of threads and variables against the safety criteria of Scott [11], which are stronger than opacity.

2 Framework

We describe a framework to express transactions and their correctness properties.

Preliminaries. Let V be a set $\{1, \ldots, k\}$ of k *variables*, and let $C = \{\mathsf{commit}\} \cup (\{\mathsf{read}, \mathsf{write}\} \times V)$ be the set of *commands* on the variables V. Also, let $\hat{C} = C \cup \{\mathsf{abort}\}$. Let $T = \{1, \ldots, n\}$ be a set of n *threads*. Let $\hat{S} = \hat{C} \times T$ be the set of *statements*. Also, let $S = C \times T$. A word $w \in \hat{S}^*$ is a finite sequence of statements. Given a word $w \in \hat{S}^*$, we define the *thread projection* $w|_t$ of w on thread $t \in T$ as the subsequence of w consisting of all statements s in w such that $s \in \hat{C} \times \{t\}$. Given a thread projection $w|_t = s_0 \ldots s_m$ of a word w on thread t, a statement s_i is *finishing in* $w|_t$ if it is a commit or an abort. A statement s_i is *initiating in* $w|_t$ if it is the first statement in $w|_t$, or the previous statement s_{i-1} is a finishing statement.

Transactions. Given a thread projection $w|_t$ of a word w on thread t, a consecutive subsequence $x = s_0 \ldots s_m$ of $w|_t$ is a *transaction* of thread t in w if (i) s_0 is initiating in $w|_t$, and (ii) s_m is either finishing in $w|_t$, or s_m is the last statement in $w|_t$, and (iii) no other statement in x is finishing in $w|_t$. The transaction x is *committing* in w if s_m is a commit. The transaction x is *aborting* in w if s_m is an abort. Otherwise, the transaction x is *unfinished* in w. Given a word w and two transactions x and y in w (possibly of different threads), we say that x *precedes* y in w, written as $x <_w y$, if the last statement of x occurs before the first statement of y in w. A word w is *sequential* if for every pair x, y of transactions in w, either $x <_w y$ or $y <_w x$. We define a function $com : \hat{S}^* \to S^*$ such that for all words $w \in \hat{S}^*$, the word $com(w)$ is the subsequence of w which consists of every statement in w that is a part of a committing transaction. A transaction x of a

thread t *writes* to a variable v if x contains a statement $((\mathsf{write}, v), t)$. A statement $s = ((\mathsf{read}, v), t)$ in x is a *global read* of a variable v if there is no statement $((\mathsf{write}, v), t)$ before s in the transaction x. A transaction x of a thread t *globally reads* a variable v if there exists a global read of variable v in transaction x.

Correctness properties. We consider two correctness properties for transactional memories: strict serializability and opacity. Strict serializability [9] requires that the order of conflicting statements from committing transactions is preserved, and the order of non-overlapping transactions is preserved. Opacity, in addition to strict serializability, requires that even aborting transactions do not read inconsistent values. The motivation behind the stricter requirement for aborting transactions in opacity is that in STMs, inconsistent reads may have unexpected side effects, like infinite loops, or array bound violations.

A statement s_1 of transaction x and a statement s_2 of transaction y (where x is different from y) *conflict* in a word w if (i) s_1 is a global read of some variable v, and s_2 is a commit, and y writes to v, or (ii) s_1 and s_2 are both commits, and x and y write to some variable v. This notion of conflict corresponds to the deferred update semantics [8] in transactional memories, where the writes of a transaction are made global upon the commit. A word $w = s_0 \dots s_m$ is *strictly equivalent* to a word w' if (i) for every thread $t \in T$, we have $w|_t = w'|_t$, and (ii) for every pair s_i, s_j of statements in w, if s_i and s_j conflict and $i < j$, then s_i occurs before s_j in w', and (iii) for every pair x, y of transactions in w, where x is a committing or an aborting transaction, if $x <_w y$, then it is not the case that $y <_{w'} x$. We define the correctness property *strict serializability* $\pi_{ss} \subseteq \hat{S}^*$ as the set of words w such that there exists a sequential word w', where w' is strictly equivalent to $com(w)$. Furthermore, we define *opacity* $\pi_{op} \subseteq \hat{S}^*$ as the set of words w such that there exists a sequential word w', where w' is strictly equivalent to w. We note that $\pi_{op} \subseteq \pi_{ss}$, that is, if a word is opaque, then it is strictly serializable.

3 Transactional Memory Specifications

We capture correctness properties using TM specification automata. A *transition system* is a 3-tuple $\langle Q, q_{init}, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, and $\delta \subseteq Q \times \hat{S} \times Q$ is a transition relation. A transition system is *deterministic* if for every state $q \in Q$ and every statement $s \in \hat{S}$, there is at most one state $q' \in Q$ such that $(q, s, q') \in \delta$. A word $s_0 \dots s_m$ is a *run* of the transition system if there exist states $q_0 \dots q_{m+1}$ in Q such that $q_0 = q_{init}$ and for all i such that $0 \leq i \leq m$, we have $(q_i, s_i, q_{i+1}) \in \delta$. The *language* L of a transition system is the set of all runs of the transition system. A *TM specification* Σ for a correctness property π is a transition system such that $L(\Sigma) = \pi$. A TM specification is *deterministic* if it is a deterministic transition system.

Strict serializability and opacity have been formally defined so far using nondeterministic TM specifications [4]. The nondeterminism allows a natural construction of the specification, where a transaction nondeterministically guesses a serialization point during its lifetime. A branch of the nondeterministic specification corresponds to a specific serialization choice of the transactions, which

makes the construction simple and intuitive, though redundant. Due to the nondeterminism of the specification, the existence of a simulation relation is a *sufficient but not a necessary* condition for language containment. This makes the decision procedure incomplete [4]. Moreover, these specifications are too large to be determinized automatically.

3.1 Difficulties in Providing Deterministic TM Specifications

It turns out that creating deterministic TM specifications for strict serializability and opacity is a non-trivial problem. We first give some examples that manifest the subtleties involved.

Analysis of strict serializability. We look at two words and reason whether they are strictly serializable.

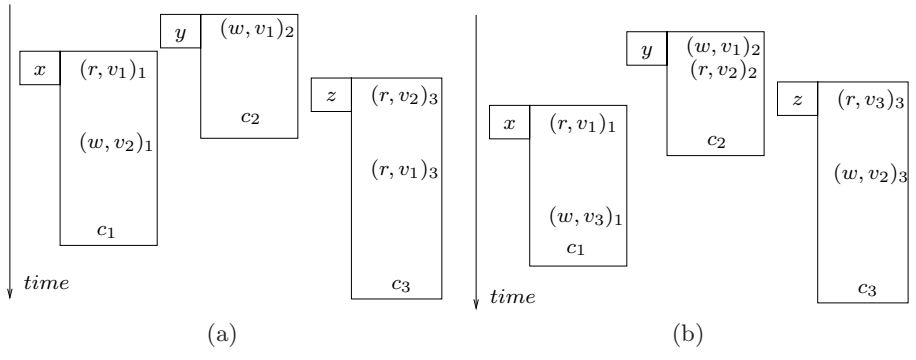


Fig. 1. Examples for strict serializability. The words are fragmented into transactions of different threads. We use the notation: w for write, r for read, c for commit, and a for abort.

- Consider the word $w = ((\mathsf{write}, v_1), t_2), ((\mathsf{read}, v_1), t_1), ((\mathsf{read}, v_2), t_3), (\mathsf{commit}, t_2), ((\mathsf{write}, v_2), t_1), ((\mathsf{read}, v_1), t_3), (\mathsf{commit}, t_1), (\mathsf{commit}, t_3)$. The word w is illustrated in Figure 1(a). The transaction x has to serialize before y due to a conflict on v_1 (as x reads v_1 before y commits and y writes to v_1). Similarly, the transaction z has to serialize before x due to a conflict on v_2. However, z has to serialize after y due to a conflict on v_1 (z reads v_1 after v_1 is written and committed by y). So, w is not strictly serializable. On the other hand, if one of the transactions had not committed, the word would have been strictly serializable.
- Consider the word $w = ((\mathsf{write}, v_1), t_2), ((\mathsf{read}, v_2), t_2), ((\mathsf{read}, v_3), t_3), ((\mathsf{read}, v_1), t_1), (\mathsf{commit}, t_2), ((\mathsf{write}, v_2), t_3), ((\mathsf{write}, v_3), t_1), (\mathsf{commit}, t_1), (\mathsf{commit}, t_3)$. The word is illustrated in Figure 1(b). The transaction x has to serialize before y due to a conflict on v_1. Similarly, the transaction z has to serialize before x due to a conflict on v_3. Also, z writes to the variable

v_2 which is read by transaction y before z commits. Thus, z has to serialize after y. This makes w not strictly serializable.

These examples show that strict serializability is a property concerned with committing transactions. Our deterministic TM specification maintains all conflicts as part of the state. We define that a transaction x is a *weak predecessor* of transaction y in a word w if y must serialize after x for both x and y to be committing transactions. When a transaction y commits, all weak predecessors of y become weak predecessors of the threads of which y is a weak predecessor. Note that the relation weak predecessor itself is not a transitive relation. The deterministic TM specification ensures that a transaction x cannot commit if x is a weak predecessor of itself. Moreover, when a transaction commits, the information of reads and writes of the transaction has to be provided to all weak predecessors of the transaction.

Analysis of opacity. Designing a deterministic specification for opacity requires even further care. This is because even aborting transactions should be prevented from reading inconsistent values. To demonstrate the intricacies involved, we again give two examples.

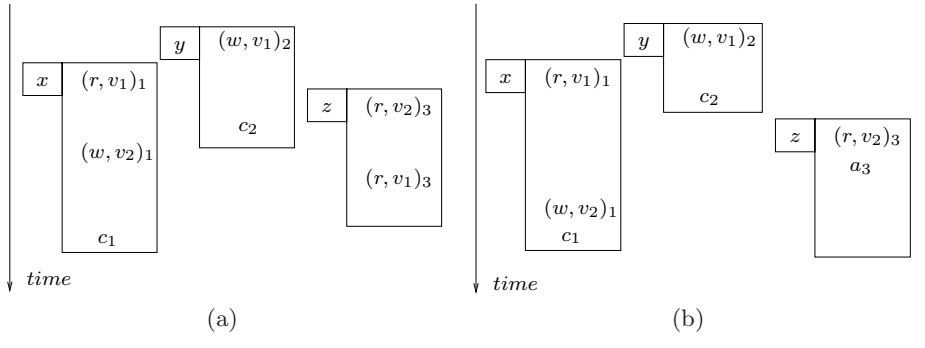


Fig. 2. Examples for opacity. The words are fragmented into transactions of different threads.

- Consider the word $w = ((\mathsf{write}, v_1), t_2), ((\mathsf{read}, v_1), t_1), ((\mathsf{read}, v_2), t_3), (\mathsf{commit}, t_2), ((\mathsf{write}, v_2), t_1), ((\mathsf{read}, v_1), t_3), (\mathsf{commit}, t_1)$. The word is illustrated in Figure 2(a). Transaction x has to serialize before y due to a conflict on v_1. Also, z has to serialize after y due to a conflict on v_1, and before x due to a conflict on v_2. Note that although z does not commit, opacity requires that transaction x does not commit. So, w is not opaque.
- Consider the word $w = ((\mathsf{write}, v_1), t_2), ((\mathsf{read}, v_1), t_1), (\mathsf{commit}, t_2), ((\mathsf{read}, v_2), t_3), (\mathsf{abort}, t_3), ((\mathsf{write}, v_2), t_1), (\mathsf{commit}, t_1)$. The word is illustrated in Figure 2(b). Transaction x has to serialize before y due to a conflict on v_1. Transaction z has to serialize after y as they do not overlap in w. Also, z

has to serialize before x due to the conflict on v_2. This makes w not opaque. This shows how a read of an aborting transaction may disallow a commit of another transaction, for the sake of opacity.

Opacity concerns committing as well as aborting transactions. Again, the deterministic TM specification for opacity maintains all conflicts as part of the state. As for strict serializability, we again use the notion of weak predecessors to store intransitive conflicts. We say that a transaction x is a *strong predecessor* of transaction y in a word w if y must serialize after x in w. Unlike weak predecessor, strong predecessor *is* a transitive relation. The specification for opacity ensures that a transaction y cannot execute *any* statement s if s makes some transaction x a strong predecessor of x. This shows how opacity poses a restriction on commands other than commit.

3.2 Deterministic TM Specifications

We now present the formal definitions of the deterministic TM specifications for strict serializability and opacity. The *deterministic TM specification for strict serializability* Σ_{ss} is given by the tuple $\langle Q, q_{init}, \delta_{ss}\rangle$. A state $q \in Q$ is a 7-tuple $\langle Status, rs, ws, prs, pws, wp, sp\rangle$, where $Status : T \to \{\mathsf{started}, \mathsf{invalid}, \mathsf{pending}, \mathsf{finished}\}$ is the status, $rs : T \to 2^V$ is the read set, $ws : T \to 2^V$ is the write set, $prs : T \to 2^V$ is the prohibited read set, $pws : T \to 2^V$ is the prohibited write set, $wp : T \to 2^T$ is the weak predecessor set, and $sp : T \to 2^T$ is the strong predecessor set for the threads. If $v \in prs(t)$ (resp. $v \in pws(t)$), then the status of the thread t is set to $\mathsf{invalid}$ if t globally reads (resp. writes to) v. A thread u is in the weak predecessor set of thread t if the unfinished transaction of u is a weak predecessor of the unfinished transaction of t. The initial state q_{init} is $\langle Status_0, rs_0, ws_0, prs_0, pws_0, wp_0, sp_0\rangle$, where $Status_0(t) = \mathsf{finished}$ for all threads $t \in T$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = wp_0(t) = sp_0(t) = \emptyset$ for all threads $t \in T$. The transition relation δ_{ss} is obtained from Algorithm 1. For all states $q \in Q$ and all statements $s \in \hat{S}$, the following hold: (i) if $specTransition(q, s, \pi_{ss}) = \bot$, then there is no state $q' \in Q$ such that $(q, s, q') \in \delta_{ss}$, and (ii) if $specTransition(q, s, \pi_{ss}) = q'$ for some state $q' \in Q$, then $(q, s, q') \in \delta_{ss}$. Given a state $q = \langle Status, rs, ws, prs, pws, wp, sp\rangle$ and a thread $t \in T$, the procedure $ResetState(q, t)$ changes $Status(t)$ to $\mathsf{finished}$ and the sets $rs(t), ws(t), prs(t), pws(t), wp(t)$, and $sp(t)$ to $\emptyset$. The deterministic TM specification for opacity builds upon the deterministic TM specification for strict serializability. The difference comes in the strong predecessor set. We exploit the relation of strong predecessors in such a way that even aborting transactions see consistent values. For example, if a thread u is a strong predecessor of t, and t is a weak predecessor of u, then u cannot commit but t can. Many similar cases of conflict have to be carefully considered to capture the exact notion of opacity, that is, $L(\Sigma_{op}) = \pi_{op}$. The *deterministic TM specification for opacity* Σ_{op} is given by the tuple $\langle Q, q_{init}, \delta_{op}\rangle$. The set of states and the initial state are the same as those for Σ_{ss}. Also, the transition relation δ_{op} can be similarly obtained from Algorithm 1 using the property π_{op} in place of π_{ss}.

Algorithm 1. *specTransition*$(\langle Status, rs, ws, prs, pws, wp, sp\rangle, s, \pi)$

if $s = ((\mathsf{read}, v), t)$ **then**
  if $v \in ws(t)$ **then return** $\langle Status, rs, ws, prs, pws, wp, sp\rangle$
  if $\pi = \pi_{op}$ **then**
    $U := \{u \in T \mid v \in prs(u)$ or $v \in prs(u')$ such that $u \in sp(u')\}$
    if $t \in U$ or there exists a thread $u \in U$ such that $t \in sp(u)$ **then return** $\bot$
  if $Status(t) = \mathsf{finished}$ **then**
    add all threads $u \in T$ such that $Status(u) = \mathsf{pending}$ to $wp(t)$ and $sp(t)$
    add all threads $u' \in T$ to $sp(t)$ such that $u' \in sp(u)$ and $Status(u) = \mathsf{pending}$
    $Status(t) := \mathsf{started}$
  $rs(t) := rs(t) \cup \{v\}$
  if $v \in prs(t)$ **then** $Status(t) := \mathsf{aborted}$
  for all threads $u \in T$ **do**
    if $v \in ws(u)$ **then** $wp(u) := wp(u) \cup \{t\}$
    if $v \in prs(u)$ **then** $wp(t) := wp(t) \cup \{u\}$
  if $\pi = \pi_{ss}$ **then return** $\langle Status, rs, ws, prs, pws, wp, sp\rangle$
  for all threads $u \in T$ such that $u = t$ or $t \in sp(u)$ **do** $sp(u) := sp(u) \cup U$
  for all threads $u \in T$ such that $u \in sp(t)$ **do**
    $pws(u) := pws(u) \cup \{v\}$
    if $v \in ws(u)$ **then** $Status(u) := \mathsf{aborted}$
if $s = ((\mathsf{write}, v), t)$ **then**
  if $Status(t) = \mathsf{finished}$ **then**
    add all threads $u \in T$ such that $Status(u) = \mathsf{pending}$ to $wp(t)$ and $sp(t)$
    add all threads $u' \in T$ to $sp(t)$ such that $u' \in sp(u)$ and $Status(u) = \mathsf{pending}$
    $Status(t) := \mathsf{started}$
  $ws(t) := ws(t) \cup \{v\}$
  if $v \in pws(t)$ **then** $Status(t) := \mathsf{aborted}$
  for all threads $u \in T$ **do**
    if $v \in rs(u)$ **then**
      $wp(t) := wp(t) \cup \{u\}$
      if $\pi = \pi_{op}$ **and** $t \in sp(u)$ **then** $Status(t) := \mathsf{aborted}$
    if $v \in pws(u)$ **then** $wp(t) := wp(t) \cup \{u\}$
if $s = (\mathsf{commit}, t)$ **then**
  if $t \in wp(t)$ **then return** $\bot$
  if $\pi = \pi_{op}$ **then**
    $U := \{u \mid u \in wp(t)$ or $u \in sp(u')$ for some $u' \in wp(t)\}$
    if $t \in U$ or there exists a thread $u \in U$ such that $t \in sp(u)$ **then return** $\bot$
  for all threads $u \in T$ such that $u \in wp(t)$ **do**
    if $ws(u) \cap ws(t) \neq \emptyset$ **then** $Status(u) := \mathsf{aborted}$ **else** $Status(u) := \mathsf{pending}$
    $prs(u) := prs(u) \cup prs(t) \cup ws(t)$
    $pws(u) := pws(u) \cup pws(t) \cup ws(t) \cup rs(t)$
    for all threads $u' \in T$ such that $t \in wp(u')$ or $ws(u') \cap ws(t) \neq \emptyset$ **do**
      $wp(u') := wp(u') \cup \{u\}$
  for all threads $u \in T$ such that $u = t$ or $t \in sp(u)$ **do** $sp(u) := sp(u) \cup U$
  ResetState(q, t)
if $s = (\mathsf{abort}, t)$ **then** *ResetState*(q, t)
return $\langle Status, rs, ws, prs, pws, wp, sp\rangle$

3.3 Model Checking with Deterministic TM Specifications

It has been shown [4] that for a transactional memory which satisfies certain structural properties, it is sufficient to show its correctness for all programs with two threads and two variables in order to prove the correctness of the transactional memory for all programs. These properties were shown for transactional memories like DSTM [6] and TL2 [2]. The nondeterministic TM specifications presented [4] are too huge to be automatically determinized. However, surprisingly enough, the deterministic TM specifications we present in this paper turn out to be much smaller in size. Using an antichain-based tool [13], we establish that for two threads and two variables, the language of our deterministic TM specification for strict serializability (resp. opacity) is equivalent to the language of the nondeterministic specification for strict serializability (resp. opacity) [4].

For strict serializability, our deterministic TM specification Σ_{ss} has only 3520 states, whereas the nondeterministic one A_{ss} has 12345 states. Similarly, for opacity, Σ_{op} has 2272 states, while the nondeterministic specification A_{op} requires 9202 states. Moreover, the deterministic TM specifications allow for an efficient procedure that directly checks, whether the language of the TM algorithm is included in the language of the deterministic TM specifications. This

Table 1. Time for simulation (resp. language inclusion) checking for STMs on a quad dual core 2.8 GHz server with 16 GB RAM. In case simulation (resp. language inclusion) holds, we write Y followed by the time required for finding it. Otherwise, we write N followed by the counterexample produced, followed by the time required to prove that no simulation exists (resp. language inclusion does not hold), followed by the time required to find the counterexample. A '*' for the search for simulation relation means that it does not complete in 2 hours, but we do find a counterexample. A '–' means that the search for both, the simulation relation and the counterexample, does not complete in 2 hours.

TM algorithm A	Number of states	$A \prec A_{ss}$	$A \prec A_{op}$	$L(A) \subseteq L(\Sigma_{ss})$	$L(A) \subseteq L(\Sigma_{op})$
Deterministic STMs [4]					
seq	3	Y, 0.8s	Y, 0.7s	Y, 0.01s	Y, 0.01s
2PL	99	Y, 13s	Y, 8s	Y, 0.01s	Y, 0.01s
dstm	944	Y, 127s	Y, 82s	Y, 0.09s	Y, 0.07s
TL2	11840	Y, 1647s	Y, 1438s	Y, 1.2s	Y, 1s
occ	4480	Y, 765s	N, w_1, 567s,4s	Y, 0.46s	N, w_1, 0.41s, 4s
TL2 mod.	17520	N, w_2, *, 9s	N, w_2, *, 9s	N, w_2, 2.7s, 9s	N, w_2, 2.1s, 8s
Nondeterministic STMs					
dstm	1846	Y, 303s	Y, 279s	Y, 0.16s	Y, 0.13s
TL2	21568	–	–	Y, 3.2s	Y, 2.4s
Counterexamples					
w_1	$(w,1)_2, (r,1)_1, c_2, (r,1)_1$				
w_2	$(w,2)_1, (w,1)_2, (r,2)_2, (r,1)_1, c_2, c_1$				

procedure makes our model checking complete too. We show the results in Table 1. For deterministic STMs [4], we observe that checking language inclusion with deterministic TM specifications is much faster than checking existence of a simulation relation with nondeterministic TM specifications.

4 Nondeterministic Transactional Memories

Our succinct deterministic TM specifications tempt us to go a step further in model checking transactional memories. Transactional memories often employ nondeterministic schemes to resolve conflicts, in the face of thread failures or repetitive aborts of a thread. These schemes are generally treated externally to the transactional memory, and are referred to as contention managers. The notion of a contention manager helps to keep the design of a transactional memory modular. This allows a transactional memory to switch from one contention manager to another, depending upon the contention scenario [5]. An STM is designed in such a way that it maintains its correctness property for all possible contention managers.

Transactional memories have been modeled in a restrictive framework as TM algorithms [4], where a transactional memory is tied to an implicit, *specific* contention manager. We now give a general formalism which is practically more useful, where a transactional memory is separated from the contention manager.

4.1 A Formalism for TM with Contention Managers

Programs. We express a thread program as an infinite binary trees on commands. For every command of a thread, we define two successor commands, one if the command is successfully executed, and another if the command fails due to an abort of the transaction. We use a set of thread programs to define a multithreaded program. Formally, a *thread program* θ on a set C of commands is a function $\theta : \mathbb{B}^* \to C$. We define a *(multithreaded) program* p on n threads and k variables as an n-tuple $p = \langle \theta^1, \ldots, \theta^n \rangle$ of thread programs on C.

TM algorithms. We model transactional memories using TM algorithms. A TM algorithm consists of a set of states, an initial state, an extended set of commands depending on the underlying TM, a conflict function, a pending function, and a transition relation between the states. The extended commands include the set C of commands, and TM specific additional commands. For example, a given TM may require that a thread locks a variable before writing to the variable. Every extended command is assumed to execute atomically. The conflict function captures the statements in the states, when the TM algorithm needs to consult a contention manager for a decision. The pending function represents the pending command of a thread in a state, and ensures that if a thread has not finished the execution of a particular command, then no other command is executed by the thread.

We define a *TM algorithm* $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, D is the set of extended commands with $C \subseteq D$, $\phi : Q \times D \to \mathbb{B}$ is the conflict function, $\gamma : Q \times T \to C \cup \{\bot\}$ is the pending function, and $\delta \subseteq Q \times \hat{C} \times \hat{S}_D \times Resp \times Q$ is the transition relation, where $\hat{S}_D = (D \cup \{\mathsf{abort}\}) \times T$ and $Resp = \{\bot, 0, 1\}$. For a TM algorithm $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$, the following rules hold:

- For all threads $t \in T$, we have $\gamma(q_{init}, t) = \bot$.
- For all states $q, q' \in Q$ such that there is an incoming transition $(q, c, (d, t), r, q')$ to q' in δ, if $r = \bot$, then $\gamma(q', t) = c$, otherwise $\gamma(q', t) = \bot$.
- For all states $q, q' \in Q$ such that there is an incoming transition $(q, c, (d, t), r, q')$ to q' in δ, then $\gamma(q', u) = \gamma(q, u)$ for all threads $u \neq t$.
- For all states q and all threads t, if $\gamma(q, t) = c$ where $c \neq \bot$, then for all outgoing transitions $(q, c_1, (d, t), r, q')$ from q in δ, we have $c_1 = c$.
- For all states q and all threads t, if $\gamma(q, t) = \bot$, then there is an outgoing transition $(q, c, (d, t), r, q'$ from q in δ for every command $c \in C$.
- For all $q \in Q$, for all transitions $(q, c, (d, t), r, q')$ in δ, we have $d = \mathsf{abort}$ if and only if $r = 0$.

Note that the rules above restrict the transition relation δ and the pending function γ such that γ is unique. A command c is *enabled* in a state q for thread t if $\gamma(q, t) \in \{\bot, c\}$ (i.e., either no command is pending, or c itself is pending). A command c is *abort enabled* in a state q for thread t if c is enabled in q for thread t and there is no transition $(q, c, (d, t), r, q') \in \delta$ such that $d \in D$. A transition relation δ is *deterministic* if for all $q \in Q$ and $(c, t) \in S$, if $(q, c, (d_1, t), r_1, q_1) \in \delta$ and $(q, c, (d_2, t), r_2, q_2) \in \delta$, then $d_1 = d_2$, $r_1 = r_2$, and $q_1 = q_2$. A TM algorithm is *deterministic* if its transition relation is deterministic.

Contention managers. When the transactional memory detects a conflict (the conflict function is true), it requests the contention manager to resolve the conflict. The contention manager proposes the TM algorithm the next statement to be executed. Formally, a *contention manager cm* on a set D of commands is a function $cm : \hat{S}_D^* \to 2^{\hat{S}_D}$, such that if the last statement of w is from thread t, then every statement in $cm(w)$ is a statement of t.

Given a TM algorithm $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$ and a contention manager $cm : \hat{S}_D^* \to 2^{\hat{S}_D}$, we define a *TM algorithm and contention manager pair* $\langle M, cm \rangle = \langle Q_\times, (q_{init}, \varepsilon), D, \gamma_\times, \delta_\times \rangle$, where $Q_\times = Q \times \hat{S}_D^*$ is the set of states, $\gamma_\times : Q_\times \times T \to C \cup \{\bot\}$ is the pending function such that for all states $q_\times \in Q_\times$ and all threads $t \in T$, we have $\gamma_\times(q_\times, t) = \gamma(q, t)$ where $q_\times = (q, w)$ for some word $w \in \hat{S}_D^*$, $\delta_\times \subseteq Q_\times \times \hat{C} \times \hat{S}_D \times Resp \times Q_\times$ is the transition relation such that for all states $q_\times, q'_\times \in Q_\times$, for all commands $c \in \hat{C}$, for all statements $s \in \hat{S}_D$, and for all responses $r \in Resp$, we have $(q_\times, c, s, r, q'_\times) \in \delta_\times$ if and only if (i) there is a transition $(q, c, s, r, q') \in \delta$, and (ii) if $\phi(q, s) = \mathsf{true}$, then $s \in cm(w)$, where $w \in \hat{S}_D^*$ and $q, q' \in Q$ such that $q_\times = (q, w)$ and $q'_\times = (q', w \cdot s)$.

Runs and languages of TM algorithms. On putting the pieces together, a TM algorithm interacts with a program, a scheduler, and a contention manager (see Fig. 3). A thread of the program is chosen by the scheduler, and the next

command of the thread is given to the TM algorithm. The TM algorithm decides whether the command can be executed in a single or several atomic steps, or the command is in conflict. The commands executed by the TM algorithm are also reported to the contention manager for its bookkeeping. If the TM algorithm finds a conflict, the TM algorithm resolves the conflict using the contention manager. The TM algorithm makes a transition accordingly, and gives back to the program a response. The response is $\bot$ if the TM algorithm needs additional steps to complete the command, 0 if the TM algorithm needs to abort the transaction of the scheduled thread, and 1 if the TM algorithm has completed the command. Given a program, a scheduler, a TM algorithm, and a contention manager, we get a run. Projecting the run to the set of successful statements (that is, aborts, and statements that get response 1) gives an infinite word. The language of a TM algorithm and contention manager pair is the set of infinite words that the TM algorithm can produce for any program and any scheduler, where conflicts are resolved using the specific contention manager.

Formally, a *scheduler* σ on T is a function $\sigma : \mathbb{N} \to T$. Let $p = \langle \theta^1, \ldots, \theta^n \rangle$ be a program, and let σ be a scheduler. A *run* $\rho = \langle q_0, l_0, (d_0, t_0), r_0 \rangle \langle q_1, l_1, (d_1, t_1), r_1 \rangle \ldots$ of a TM algorithm A with scheduler σ on program p and contention manager cm is an infinite sequence of tuples of states, program locations, statements, and responses, where $l_j = \langle l_j^1, \ldots, l_j^n \rangle \in (\mathbb{B}^*)^n$ for all $j \geq 0$ and the following hold: (i) $q_0 = q_{init}$ and $l_0 = \langle \varepsilon, \ldots, \varepsilon \rangle$, and (ii) for all $j \geq 0$, there exists a transition $(q_j, c_j, (d_j, t_j), r_j, q_{j+1}) \in \delta$ such that if $\phi(q_j, (d_j, t_j)) = \mathsf{true}$, then $(d_j, t_j) \in cm((d_0, t_0) \ldots (d_{j-1}, t_{j-1}])$, and (iii) $t_j = \sigma(j)$, and (iv) $c_j = \theta^{t_j}(l_j^{t_j})$, and (v) for all $t \in T$, we have $l_{j+1}^t = l_j^t$ if either $t \neq t_j$ or $r_j = \bot$, and $l_{j+1}^t = l_j^t \cdot r_j$ otherwise. A statement $s_j \in \hat{S}$ is *successful* in the run $\rho = \langle q_0, l_0, s_0, r_0 \rangle \langle q_1, l_1, s_1, r_1 \rangle \ldots$ if (i) $r_j \in \{0, 1\}$, or (ii) $r_k = 1$ with $j < k$ and $r_{j+1} \ldots r_{k-1}$ are all equal to $\bot$. We define the *language* $L(\langle A, cm \rangle)$ of a $\langle A, cm \rangle$ pair as the set of all infinite words $w \in \hat{S}^\omega$ such that w is the sequence of all successful statements in a run of A with some scheduler on some program and the contention manager cm. A TM algorithm A with a contention manager cm ensures a correctness property $\pi \subseteq \hat{S}^*$ if every finite prefix of every word in $L(\langle A, cm \rangle)$ is in π.

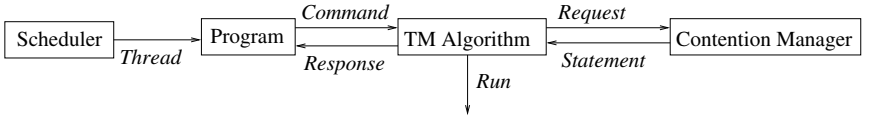


Fig. 3. Interaction in the model

Modeling contention managers explicitly in our formalism is not a feasible option. First of all, contention managers may blow up the state space as their decisions may depend intricately on past behavior. For example, a simple random backoff contention manager, that asks a conflicting thread to back off for a random amount of time could blow up the state space. Secondly, some of the structural properties break when we model a TM algorithm in conjunction with

a particular contention manager. For example, if a contention manager prioritizes transactions according to the number of times it has aborted in the past, then the TM algorithm does not satisfy the structural property of 'transactional projection' [4]. This is because, an abort of a transaction of thread t may be the reason why the next transaction of thread t commits. As the remaining structural properties build upon the transactional projection property, they also collapse for specific contention managers.

We take a novel approach to model check transactional memories with different contention managers. Given a TM algorithm A with extended alphabet D, we define a *universal contention manager ucm* such that for all words $w \in \hat{S}^*_D$, we have $ucm(w) = \hat{S}_D$. The idea of the universal contention manager is to allow nondeterministically all choices that the TM algorithm has. It is easy to observe that the transition relation for the pair $\langle A, ucm\rangle$ is identical to that of the TM algorithm A. From the definition of the language of a TM algorithm and a contention manager pair, we get $L(\langle A, cm\rangle) \subseteq L(\langle A, ucm\rangle)$ for every contention manager cm. Thus, if a TM algorithm ensures a correctness property with the universal contention manager, then the TM algorithm is correct *for all* contention managers. Moreover, if a TM algorithm A satisfies the structural properties, then the pair $\langle A, ucm\rangle$ also satisfies the structural properties [4]. Thus, verifying the correctness of the TM algorithm with ucm for two threads and two variables proves the correctness of the TM algorithm for arbitrary number of threads and variables for all possible contention managers.

We now provide, as examples, nondeterministic DSTM and nondeterministic TL2, combined with the universal contention manager. We then verify their correctness.

4.2 Nondeterministic DSTM

Dynamic software transactional memory (DSTM) [6] is one of the most popular transactional memories. DSTM faces a conflict when a transaction wants to own a variable which is owned by another thread. We define the nondeterministic DSTM algorithm A_{dstm} as $\langle Q, q_{init}, D, \gamma, \delta_{dstm}\rangle$. A state $q \in Q$ is defined as a 3-tuple $\langle Status, rs, os\rangle$, where $Status : T \to \{\mathsf{aborted}, \mathsf{validated}, \mathsf{invalid}, \mathsf{finished}\}$ is the status function, and $rs : T \to V$ is the read set, and $os : T \to V$ is the ownership set.

The initial state $q_{init} = \langle Status_0, rs_0, os_0\rangle$, where for all threads $t \in T$, we have $Status_0(t) = \mathsf{finished}$ and $rs_0(t) = os_0(t) = \emptyset$. The set of extended commands is $D = C \cup (\{\mathsf{own}\} \times V) \cup \{\mathsf{validate}\}$. The transition relation δ_{dstm} is obtained from Algorithm 2. For all states $q \in Q$, all commands $c \in C$, all extended commands $d \in D \cup \{\mathsf{abort}\}$, all threads $t \in T$, and all responses $r \in Resp$, we have: (i) if $dstmTransition(q, c, d, t, r) = \bot$, then there does not exist a state $q' \in Q$ such that $(q, c, (d, t), r, q') \in \delta_{dstm}$, and (ii) if $dstmTransition(q, c, d, t, r) = q'$ for some state $q' \in Q$, then $(q, c, (d, t), r, q') \in \delta_{dstm}$.

Our second example is a model of another popular transactional memory, transactional locking 2 (TL2) [2] with the universal contention manager. We give an informal description of the role of ucm in TL2. TL2 uses locks for ensuring

Algorithm 2. *dstmTransition*$(\langle Status, rs, os\rangle, c, d, t, r)$

if c is not enabled in q for thread t **then return** $\bot$
if $c = (\mathsf{read}, v)$ **then**
  if $d = c$ **and** $v \in os(t)$ **and** $r = 1$ **and** $Status(t) \neq \mathsf{aborted}$ **then return** q
  if $d = c$ **and** $v \notin os(t)$ **and** $r = 1$ **and** $Status(t) = \mathsf{finished}$ **then**
    $rs(t) := rs(t) \cup \{v\}$
    return q
if $c = (\mathsf{write}, v)$ **then**
  if $d = c$ **and** $v \in os(t)$ **and** $r = 1$ **and** $Status(t) \neq \mathsf{aborted}$ **then return** q
  if $d = (\mathsf{own}, v)$ **and** $r = \bot$ **and** $Status(t) \neq \mathsf{aborted}$ **then**
    $os(t) := os(t) \cup \{v\}$
    for all threads $u \neq t$ such that $v \in os(u)$ **do**
      $Status(u) := \mathsf{aborted} \quad rs(u) := \emptyset \quad os(u) := \emptyset$
    return q
if $c = \mathsf{commit}$ **then**
  if $d = \mathsf{validate}$ **and** $r = \bot$ **and** $Status(t) = \mathsf{finished}$ **then**
    $Status(t) := \mathsf{validated}$
    for all threads $u \neq t$ such that $rs(t) \cap os(u) \neq \emptyset$ **do**
      $Status(u) := \mathsf{aborted} \quad rs(u) := \emptyset \quad os(u) := \emptyset$
    return q
  if $d = c$ **and** $r = 1$ **and** $Status(t) = \mathsf{validated}$ **then**
    $Status(t) := \mathsf{finished} \quad rs(t) := \emptyset \quad os(t) := \emptyset$
    for all threads $u \neq t$ such that $rs(u) \cap os(t) \neq \emptyset$ **do** $Status(u) := \mathsf{invalid}$
    return q
if $d = \mathsf{abort}$ **and** $r = 0$ **then**
  $Status(t) := \mathsf{finished} \quad rs(t) := \emptyset \quad os(t) := \emptyset$
  if c is abort enabled in q **and** $d = \mathsf{abort}$ **and** $r = 0$ **then return** q
  if $c = (\mathsf{write}, v)$ **and** $v \notin os(t)$ **and** $v \in os(u)$ s.t. $u \neq t$ **then return** q
  if $c = \mathsf{commit}$ **and** $Status(t) = \mathsf{finished}$ **and** $rs(t) \cap os(u) \neq \emptyset$ s.t. $u \neq t$ **then**
    return q
return $\bot$

opacity. A thread locks all the variables in the write set at the time of commit. With TL2 algorithm using the universal contention manager, whenever a thread t conflicts due to a variable being locked by another thread u, the nondeterministic TL2 algorithm has the following transitions: one to abort t, and others to allow the thread t to proceed by setting the abort flag of some thread u.

We note that nondeterministic DSTM and nondeterministic TL2, combined with the universal contention manager satisfy the transactional projection property, as aborting or unfinished transactions can influence committing transactions only by forcing them to abort. The remaining structural properties depend on the transactional projection property, but are not influenced by a contention manager. Thus, all required structural properties do hold for nondeterministic DSTM and nondeterministic TL2 obtained with the universal contention manager. We check whether the language of these nondeterministic STMs is included in the language of the deterministic TM specifications. Our results, shown in Table 1, establish that DSTM and TL2 ensure opacity for an arbitrary number

of threads and variables for all contention managers. We observe that the number of states in the nondeterministic TM algorithm using the universal contention manager is nearly double the number of states in the corresponding deterministic TM algorithm. We note that the nondeterministic specifications are unable to verify the correctness properties for the nondeterministic TL2 algorithm.

5 Conclusion

We presented deterministic specifications for two key correctness properties, strict serializability and opacity, in transactional memories. Our deterministic specifications make the model checking procedure for transactional memories complete and efficient. We formalized the notion of nondeterministic transactional memories to capture realistic contention management. We proved that DSTM and TL2 ensure opacity with arbitrary numbers of threads and variables for all possible contention managers.

Acknowledgment. We are thankful to Laurent Doyen for his kind support in checking language inclusion with his antichain based tool.

References

1. Cohen, A., O'Leary, J., Pnueli, A., Tuttle, M.R., Zuck, L.: Verifying correctness of transactional memories. In: FMCAD, pp. 37–44 (2007)
2. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
3. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP, pp. 175–184 (2008)
4. Guerraoui, R., Henzinger, T.A., Jobstmann, B., Singh, V.: Model checking transactional memories. In: PLDI (2008)
5. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 303–323. Springer, Heidelberg (2005)
6. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101 (2003)
7. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA, pp. 289–300. ACM Press, New York (1993)
8. Larus, J.R., Rajwar, R.: Transactional Memory. Synthesis Lectures on Computer Architecture. Morgan & Claypool (2007)
9. Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM, 631–653 (1979)
10. Scherer, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC, pp. 240–248 (2005)
11. Scott, M.L.: Sequential specification of transactional memory semantics. In: TRANSACT (2006)
12. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
13. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)

Semantics of Deterministic Shared-Memory Systems*

Rémi Morin

Aix-Marseille université — UMR 6166 — CNRS
Laboratoire d'Informatique Fondamentale de Marseille
163, avenue de Luminy, F-13288 Marseille Cedex 9, France

Abstract. We investigate a general model of concurrency for shared-memory systems. We introduce some intuitive interleaving semantics within the general framework of automata with concurrency relations and connect it to some partial order approach. Then our main result identifies the expressive power of finite deterministic shared-memory systems with the notion of regular consistent sets of labeled partial orders. We characterize also by means of a coherence property the languages recognized by deadlock-free systems.

Introduction

The concurrent executions of Petri nets or asynchronous systems, and more generally Mazurkiewicz traces, can be regarded as labeled partial orders [4, 6, 19]. Besides other models of distributed systems such as message-passing systems are provided with a partial order view of their executions called message sequence charts [12]. In this paper we investigate a general model for shared-memory systems and we show that such systems can be given a natural partial order semantics as well. We will observe that these systems are a generalization of 1-safe Petri nets [19], asynchronous automata [24], and asynchronous cellular automata [5]. To a certain extent this model subsumes the framework of channel-bounded message-passing systems [12], too.

Basically the partial order approach of concurrent executions that we adopt respects the following point of view: *Two events must be ordered if they occur on the same process or if one event reads the value or writes a new value in a register that is also modified by the other*. In other words we consider Concurrent-Read-Exclusive-Write systems. This point of view is actually a simple generalization of the way events are ordered in the restricted case of Mazurkiewicz traces and asynchronous automata. The variant of asynchronous *cellular* automata models a kind of shared-memory systems where each process communicates with a fixed subset of neighbors. In this paper we study a more general model where the communication connectivity evolves dynamically along executions. As a result the labeled partial orders associated with these shared-memory systems are no longer Mazurkiewicz traces. Still our approach differs from the setting of [1] and [9] which adopt a more relaxed notion of dependency.

The analysis of a distributed protocol is often easier to understand with the visual description of the interactions between processes and the causality between events by means of a partial order. For instance Peterson's mutual exclusion protocol for two processes can be formalized by the automaton from Figure 1 and a typical execution

* Supported by the ANR project SOAPDC.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 36–51, 2008.

of this system is depicted in the top-down usual way by the labeled partial order from Fig. 2. In Section 1 we introduce a partial order semantics of shared-memory systems based on the formalization of the May-Occur-Concurrently relation between transitions rules and the Must-Happen-Before relation between occurrences of actions.

Mazurkiewicz traces are labeled partial orders that arise in a natural manner from partial commutations of actions [4, 6]. In that way the intuitive interleaving of actions along an execution corresponds exactly to a particular labeled partial order. A similar duality appears with message sequence charts which can be regarded as labeled partial orders or equivalence classes of words with respect to some configuration-dependent independence relation [12]. We show in Section 2 that our partial order view of the executions of a shared-memory system corresponds to a natural interleaving approach based on the notion of automata with concurrency relations [8].

The concurrent executions of some finite deterministic asynchronous automaton form a regular set of Mazurkiewicz traces. Zielonka's celebrated theorem asserts the converse property [5, 24]: Any regular set of Mazurkiewicz traces is accepted by some finite deterministic asynchronous automaton. A similar relationship holds between regular sets of message sequence charts and finite deterministic message-passing systems [12]. Both connections admit also a variant that characterizes which regular languages can be accepted by some *deadlock-free* systems [3, 18, 22]. We establish here similar relationships in Corollaries 4.1 and 4.2 between finite deterministic shared-memory systems and regular consistent sets of pomsets, a notion borrowed from [2].

Preliminaries. A labeled partial order or *pomset* (for partially ordered multiset) over an alphabet Σ is a triple $t = (E, \preccurlyeq, \xi)$ where $(E, \preccurlyeq)$ is a finite partial order and ξ is a mapping from E to Σ *without autoconcurrency*: $\xi(x) = \xi(y)$ implies $x \preccurlyeq y$ or $y \preccurlyeq x$ for all $x, y \in E$. We denote by $\mathbb{P}(\Sigma)$ the class of all pomsets over Σ. A pomset can be seen as an abstraction of an execution of a concurrent system [6, 14, 19, 20]. In this view, the elements x of E are *events* and their label $\xi(x)$ describes the action performed when event x occurs. Moreover the ordering $x \preccurlyeq y$ means that x *must happen before* y.

Let $t = (E, \preccurlyeq, \xi)$ be a pomset and $x, y \in E$. Then y *covers* x (denoted $x \multimap y$) if $x \prec y$ and $x \prec z \preccurlyeq y$ implies $y = z$. An *order extension* of a pomset $t = (E, \preccurlyeq, \xi)$ is a pomset $t' = (E, \preccurlyeq', \xi)$ such that $\preccurlyeq \subseteq \preccurlyeq'$. A *linear extension* of t is an order extension that is linearly ordered. It corresponds to a sequential view of the concurrent execution t. Linear extensions of a pomset t over Σ can naturally be regarded as words over Σ. By $\mathrm{LE}(t) \subseteq \Sigma^\star$, we denote the set of linear extensions of a pomset t over Σ. For any subset of pomsets $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$, we put $\mathrm{LE}(\mathcal{L}) = \bigcup_{t \in \mathcal{L}} \mathrm{LE}(t)$.

Two isomorphic pomsets admit the same set of linear extensions. Noteworthy the converse property holds [23]: If $\mathrm{LE}(t) = \mathrm{LE}(t')$ then t and t' are two isomorphic pomsets. In the sequel of this paper we do not distinguish between isomorphic pomsets any longer because they are used as representative of sets of words. In particular, $\mathrm{LE}(t) = \mathrm{LE}(t')$ implies $t = t'$.

An *ideal* of a pomset $t = (E, \preccurlyeq, \xi)$ is a subset $H \subseteq E$ such that $x \in H$ and $y \preccurlyeq x$ imply $y \in H$. The restriction $t|H = (H, \preccurlyeq \cap (H \times H), \xi \cap (H \times \Sigma))$ is called a *prefix* of t and we write $t' \leqslant t$. For all $z \in E$, we denote by $\downarrow z$ the ideal of events below z, i.e. $\downarrow z = \{y \in E \mid y \preccurlyeq z\}$. For any set of pomsets $\mathcal{L}$, $\mathrm{Pref}(\mathcal{L})$ denotes the set of prefixes of pomsets from $\mathcal{L}$. We say that $\mathcal{L}$ is *prefix-closed* if $\mathrm{Pref}(\mathcal{L}) = \mathcal{L}$.

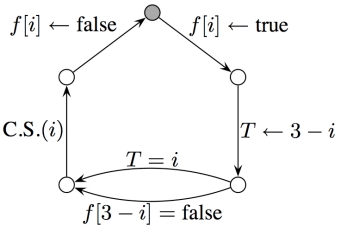


Fig. 1. Process i of Peterson's protocol

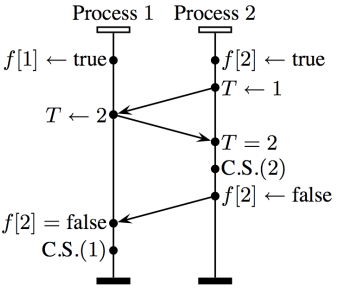


Fig. 2. A sample scenario

1 A General Model for Communicating Systems

Throughout the paper we fix some (possibly infinite) alphabet Σ. The notion of a shared-memory system we consider is based on a set $\mathcal{I}$ of processes together with a distribution $\mathrm{Loc} : \Sigma \to 2^{\mathcal{I}}$ which assigns to each $a \in \Sigma$ a fixed subset of processes $\mathrm{Loc}(a) \subseteq \mathcal{I}$. Intuitively each occurrence of action a induces a *synchronized* step of all processes from $\mathrm{Loc}(a)$. For that reason we assume that $\mathrm{Loc}(a)$ is non-empty for all $a \in \Sigma$. In many examples, such as safe Petri nets [19] and asynchronous cellular automata [5, 9], each action turns out to occur on a single process so that processes never synchronize and the process alphabets $\mathrm{Loc}^{-1}(\{i\}) \subseteq \Sigma$ are disjoint. Still in this paper we allow processes to share actions in order to take the classical models of asynchronous automata [24] and mixed product of automata [10] into account.

1.1 Shared-Memory Systems

Processes of a shared-memory system can communicate by means of a set $\mathcal{R}$ of shared variables (or registers) taking values from a common set of data $\mathcal{D}$; in particular the initial contents of this shared memory is formalized by a *memory-state* $\chi_{\text{init}} : \mathcal{R} \to \mathcal{D}$ that associates to each register $r \in \mathcal{R}$ a value $\chi_{\text{init}}(r) \in \mathcal{D}$. Intuitively each action corresponds to the reading of the values of a subset of registers (a guard) and the writing of new values in some other registers. For convenience, we shall allow a concurrent reading of the value of a register by distinct processes; but we forbid the writing of a new value in the same register by two different processes simultaneously, that is, we shall consider *Concurrent-Read Exclusive-Write* systems, only. A *valuation* is a partial function $\nu : \mathcal{R} \rightharpoonup \mathcal{D}$; it will correspond to the reading or the writing of some values in a subset of registers. The *domain* $\mathrm{dom}(\nu)$ of a valuation ν is the set of registers r such that $\nu(r)$ is defined. We denote by $\mathcal{V}$ the set of all valuations.

Now each process $i \in \mathcal{I}$ is provided with a set of local states S_i together with an initial local state $\imath_i \in S_i$. A *global state* $s = (s_i)_{i \in \mathcal{I}}$ consists of one local state s_i for each process $i \in \mathcal{I}$ and a *configuration* $q = (\chi, s)$ is a pair made of a memory-state $\chi : \mathcal{R} \to \mathcal{D}$ and a global state s. We let the Cartesian product $Q = \mathcal{D}^{\mathcal{R}} \times \prod_{i \in \mathcal{I}} S_i$ denote the set of all configurations. The initial configuration $\imath = (\chi_{\text{init}}, s)$ corresponds to the initial memory-state χ_{init} and the initial global state $s = (\imath_i)_{i \in \mathcal{I}}$. Given a memory-state $\chi : \mathcal{R} \to \mathcal{D}$ and a subset of registers $R \subseteq \mathcal{R}$, we let $\chi|R$ denote the valuation with

domain R such that $\chi|R(r) = \chi(r)$ for all $r \in R$. Given some action a, some process j and some global state $s = (s_i)_{i\in\mathcal{I}}$, we denote by $s|a$ the partial state $(s_i)_{i\in \mathrm{Loc}(a)}$ and by $s|j$ the local state s_j. For each $a \in \Sigma$ we denote by S_a the set of partial states $S_a = \prod_{i\in\mathrm{Loc}(a)} S_i$. A *transition rule* is a quintuple (ν, s, a, ν', s') where $a \in \Sigma$, $\nu, \nu' \in \mathcal{V}$ are two valuations and $s, s' \in S_a$ are two partial states.

Definition 1.1. *A* shared-memory system *(for short, an SMS) over some distributed alphabet* (Σ, Loc)*, some initial memory-state* $\chi_{init} : \mathcal{R} \to \mathcal{D}$*, and local states* $(S_i, \imath_i)_{i\in\mathcal{I}}$ *consists of a set of transition rules* Δ*.*

Intuitively action a can occur synchronously on all processes from $\mathrm{Loc}(a)$ in some configuration $q^\circ = (\chi^\circ, s^\circ)$ if there exists a transition rule $(\nu, s, a, \nu', s') \in \Delta$ such that $\nu = \chi^\circ|\mathrm{dom}(\nu)$ and $s = s^\circ|a$. In that case processes from $\mathrm{Loc}(a)$ may perform a joint move to the new partial state s' and write the new values $\nu'(r)$ in registers from the domain of ν'. The step consisting of all these moves and all these changes is atomic. For convenience we put $\rho = (\nu_\rho, s_\rho, a_\rho, \nu'_\rho, s'_\rho)$, $\mathrm{R}_\rho = \mathrm{dom}(\nu_\rho)$ and $\mathrm{W}_\rho = \mathrm{dom}(\nu'_\rho)$ for each transition rule ρ.

Example 1.2. Recall that a 1-safe Petri net is a structure $\mathcal{N} = (B, E, F, \mathfrak{m})$ where B is a set of conditions, E is a set of events with $E \cap B = \emptyset$, $F \subseteq (B \times E) \cup (E \times B)$ is the flow relation, and $\mathfrak{m} \subseteq B$ is an initial marking. Such a structure can be seen as an SMS where we have $\mathcal{R} = B$, $\mathcal{D} = \{0, 1\}$, $\Sigma = \mathcal{I} = E$, $\mathrm{Loc}(a) = \{a\}$, and $S_a = \{\imath_a\}$ is a singleton. Then each subset of conditions (called a marking) is identified with a memory-state $\chi : B \to \{0, 1\}$. For each event e, the flow relation defines a preset of conditions ${}^\bullet e = \{b \in B \mid (b, e) \in F\}$ and a postset of conditions $e^\bullet = \{b \in B \mid (e, b) \in F\}$. Then the transition rule $(\nu, \imath_e, e, \nu', \imath_e)$ belongs to Δ if the two next requirements are fulfilled:

- ν has domain ${}^\bullet e \cup e^\bullet$, $\nu(b) = 1$ for all $b \in {}^\bullet e$ and $\nu(b) = 0$ for all $b \in e^\bullet \setminus {}^\bullet e$;
- ν' has domain ${}^\bullet e \cup e^\bullet$, $\nu'(b) = 1$ for all $b \in e^\bullet$ and $\nu'(b) = 0$ for all $b \in {}^\bullet e \setminus e^\bullet$.

1.2 Pomset Semantics of Shared-Memory Systems

Following a classical trend in concurrency theory [14, 19, 20] we want to describe the concurrent executions of a shared-memory system $\mathcal{S}$ by means of labeled partial orders in such a way that the ordering of events represents the must-happen-before relation between occurrences of actions. Since each process works sequentially, events occurring on the same process must be comparable. Furthermore any two events that change the value of some register should be comparable, that is, we consider Exclusive-Write systems. Now if one event writes a new value in some register read by another event then these two events should be comparable as well; otherwise it would be unclear which value is actually read by the second event. In that way we have characterized which pairs of transition rules may occur concurrently. We formalize this *May-Occur-Concurrently* relation by means of a binary relation $\| \subseteq \Delta \times \Delta$. Let $\rho, \rho' \in \Delta$ be two transitions rules. We put $\rho \| \rho'$, and we say that ρ and ρ' are independent, if $\mathrm{Loc}(a_\rho) \cap \mathrm{Loc}(a_{\rho'}) = \emptyset$, $\mathrm{W}_\rho \cap (\mathrm{R}_{\rho'} \cup \mathrm{W}_{\rho'}) = \emptyset$, and $\mathrm{W}_{\rho'} \cap (\mathrm{R}_\rho \cup \mathrm{W}_\rho) = \emptyset$. Thus two transition rules are independent if they correspond to actions occurring on disjoint sets of processes and if each transition rule does not modify the registers read or written by the other.

In order to reason about which registers are read by each event and how events change the local states of processes and the values of registers, we make use of the notion of run. Let $t = (E, \preccurlyeq, \xi)$ be a pomset over Σ. A *run* of t over $\mathcal{S}$ is a mapping $\rho : E \to \Delta$ which maps each event e from E to some transition rule $\rho(e) \in \Delta$ such that $a_{\rho(e)} = \xi(e)$. In order to reflect the May-Occur-Concurrently relation, two events are incomparable in t only if their transition rules are independent. This is formalized by Axiom V_1 below. The partial order of events in t results from the transitive closure of the covering relation $\prec$ and can be represented by its Hasse diagram. Since we want the partial order to reflect the *Must-Happen-Before* relation, any edge from the covering relation must represent some dependence between the corresponding transition rules. This is formalized by Axiom V_2 below. As a consequence the run ρ is called *valid* if V_1 and V_2 are satisfied:

V_1: For all events $e_1, e_2 \in E$ with $\rho(e_1) \not\parallel \rho(e_2)$, we have $e_1 \preccurlyeq e_2$ or $e_2 \preccurlyeq e_1$;
V_2: For all events $e_1, e_2 \in E$ with $e_1 \prec e_2$, we have $\rho(e_1) \not\parallel \rho(e_2)$.

In particular if e and e' are two events that change the value of some register r then e and e' are comparable w.r.t. $\preccurlyeq$. Similarly if e and e' are two events that occur on some process $i \in \mathcal{I}$ then e and e' are comparable w.r.t. $\preccurlyeq$.

We assume now that ρ is a valid run for t. Let $H \subseteq E$ be an ideal of t. The configuration $q_{\rho,H}$ at H corresponds intuitively to a snapshot of the system after all events of H have occurred along the execution of t w.r.t. ρ: The value of each register is the value written by the last event that has modified this value and the local state of each process is the local state reached after the last joint move performed by that process. Formally $q_{\rho,H}$ is the configuration $q_{\rho,H} = (\chi_{\rho,H}, s_{\rho,H})$ defined by the next two conditions:

- For all registers $r \in \mathcal{R}$, we put $\chi_{\rho,H}(r) = \nu'_{\rho(e)}(r)$ if e is the greatest event in H such that $r \in \mathrm{W}_{\rho(e)}$, and $\chi_{\rho,H}(r) = \chi_{\text{init}}(r)$ if there is no such event.
- For all $i \in \mathcal{I}$, we put $s_{\rho,H}|i = s'_{\rho(e)}|i$ if e is the greatest event in H such that $i \in \mathrm{Loc}(\xi(e))$, and $s_{\rho,H}|i = \imath_i$ if there is no such event.

Due to V_1 events satisfying $r \in \mathrm{W}_{\rho(e)}$ are totally ordered so there exists at most one maximal event satisfying this condition. A similar observation holds for events satisfying $i \in \mathrm{Loc}(\xi(e))$. Therefore $q_{\rho,H}$ is well-defined. Note here that $q_{\rho,\emptyset}$ corresponds to the initial configuration $\imath$. Now we say that a valid run ρ is compatible with $\mathcal{S}$ if the configuration reached after all events below e enables the execution of the rule $\rho(e)$. Formally a valid run ρ of t is *compatible with* $\mathcal{S}$ if for all events $e \in E$ the configuration (χ, s) at $\downarrow e \setminus \{e\}$ satisfies $\chi|\mathrm{R}_{\rho(e)} = \nu_{\rho(e)}$ and $s|\xi(e) = s_{\rho(e)}$. A pomset that admits a compatible run corresponds to a potential execution of $\mathcal{S}$.

Definition 1.3. *A pomset over Σ is accepted by $\mathcal{S}$ if it admits a compatible run. The* language $\mathcal{L}(\mathcal{S}) \subseteq \mathbb{P}(\Sigma)$ *recognized by $\mathcal{S}$ collects all pomsets accepted by $\mathcal{S}$.*

Note that if t admits a compatible run then any prefix of t admits a compatible run, too. Therefore the pomset language $\mathcal{L}(\mathcal{S})$ is prefix-closed. We say that a configuration q is *reachable* in $\mathcal{S}$ if there exists a pomset $t \in \mathcal{L}(\mathcal{S})$ and a compatible run ρ of t such that $q = q_{\rho,E}$, that is, q describes the memory-state and the global state of the system after all events have occurred with respect to ρ.

1.3 Restriction to Deterministic Shared-Memory Systems

In the sequel of this paper we consider only deterministic shared-memory systems. Intuitively determinism means that from any reachable configuration there exists at most one transition rule that allows an occurrence of a given action.

Definition 1.4. *A shared-memory system is* deterministic *if for all actions $a \in \Sigma$ and all reachable configurations $q = (\chi, s)$ there is at most one transition rule $\rho \in \Delta$ such that $a_\rho = a$, $\nu_\rho = \chi|R_\rho$, and $s_\rho = s|a$.*

Most models of communicating systems fit into the formalism of deterministic shared-memory systems. In particular any specification in the form of a system of automata such as Peterson's protocol in Figure 1 can be formalized in this setting by considering each transition as a particular action.

1.4 Relationships with Asynchronous Automata

A second example of SMS from the literature is provided by asynchronous automata [24]. The latter correspond formally to shared-memory systems such that $\mathcal{R} = \emptyset = \mathcal{D}$. Then $\mathcal{V} = \{\emptyset\}$ and the transition rules associated with some action a form a binary relation $\delta_a \subseteq S_a \times S_a$. For deterministic systems (Def. 1.4), the latter can be regarded as a partial function $\delta_a : S_a \rightharpoonup S_a$ if we remove from δ_a all transition rules that apply only from unreachable configurations.

On the other hand the alternative definition of asynchronous automata investigated in [7, chap. 7] can be identified with the class of deterministic shared-memory systems such that $\Sigma = \mathcal{I}$ and $\mathrm{Loc}(a) = \{a\}$ for all $a \in \Sigma$, i.e. each action corresponds to a process, and for each $a \in \Sigma$, $S_a = \{\imath_a\}$ is a singleton — so there is a single global state. In this approach each action is assigned a read domain $\mathrm{R}_a \subseteq \mathcal{R}$ and a write domain $\mathrm{W}_a \subseteq \mathcal{R}$ such that $\mathrm{W}_a \subseteq \mathrm{R}_a$. It is required that $(\nu, \imath_a, a, \nu', \imath_a) \in \Delta$ holds only if ν has domain R_a and ν' has domain W_a. Due to determinism, the set of transition rules associated with a can be regarded as a partial function $\delta_a : \mathcal{D}^{\mathrm{R}_a} \rightharpoonup \mathcal{D}^{\mathrm{W}_a}$. In that way the notion of deterministic shared-memory systems we consider appears as a formal generalization of both notions of asynchronous automata. The notion of asynchronous *cellular* automata from [5, 25] also fits into our framework. These systems correspond actually to the asynchronous automata from [7] such that $\mathcal{R} = \mathcal{I}$, $\mathrm{W}_a = \{a\}$ and $b \in \mathrm{R}_a$ implies $a \in \mathrm{R}_b$ for all $a, b \in \Sigma$.

Interestingly another definition of asynchronous cellular automata was investigated in [9]. This model can be identified with a shared-memory system such that $\mathcal{R} = \mathcal{I}$, $S_i = \{\imath_i\}$ for each process $i \in \mathcal{I}$, (that is, each process owns a register whose value describes its current state), $\mathrm{Loc}(a)$ is a singleton for each action a, (so processes do not synchronize) and moreover $(\nu, \imath|a, a, \nu', \imath|a) \in \Delta$ holds only if the domain of ν' is $\{\mathrm{Loc}(a)\}$ which means that each process writes only in its own register. Such a generalized asynchronous cellular automaton is called deterministic if for all actions $a \in \Sigma$ and all valuations $\nu \in \mathcal{V}$ there exists at most one transition rule $(\nu, \imath|a, a, \nu', \imath|a) \in \Delta$. Noteworthy these deterministic generalized asynchronous cellular automata do not forbid the situation where several different (and possibly conflicting) transition rules to perform a can be applied at some configuration. For that reason this approach does not fit completely into the present setting.

2 Interleaving Semantics of Shared-Memory Systems

In this section we fix a shared-memory system $\mathcal{S}$ over Σ. We present an interleaving semantics by means of a configuration dependent independence relation and relate it to the partial order approach from Definition 1.3.

2.1 Configuration System

The *configuration system* of $\mathcal{S}$ is the transition system $\mathcal{C}(\mathcal{S}) = (Q, \imath, \Sigma, \longrightarrow)$ defined as follows: Q is the set of configurations, $\imath = (\chi_{\text{init}}, (\imath_i)_{i\in\mathcal{I}})$ is the initial configuration, and $\longrightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions such that for any two configurations $q = (\chi, s)$, $q' = (\chi', s')$ and any action a, we have $q \xrightarrow{a} q'$ if there are two subsets of registers R, W such that $(\chi|\mathrm{R}, s|a, a, \chi'|\mathrm{W}, s'|a) \in \Delta$, $s'|i = s|i$ for all $i \in \mathcal{I} \setminus \mathrm{Loc}(a)$, and $\chi'(r) = \chi(r)$ for all $r \in \mathcal{R} \setminus W$. In other words the system can evolve from q to q' by performing an action a provided that some transition rule ρ enables all processes from $\mathrm{Loc}(a)$ to proceed a joint move from $s|a$ to $s'|a$ as soon as the registers from R_ρ hold the specific values $\chi|\mathrm{R}_\rho$. Furthermore the new values $\chi'|\mathrm{W}_\rho$ are written into the registers from W_ρ in order to lead to the new configuration q'.

Example 2.1. We continue Example 1.2. Since there is a single global state, configurations can be identified with memory-states — or equivalently markings. According to the above definition, there exists a transition $\chi \xrightarrow{e} \chi'$ in the configuration system $\mathcal{C}(\mathcal{S})$ if the following requirements are fulfilled: $\chi(b) = 1$ for all $b \in {}^\bullet e$; $\chi(b) = 0$ for all $b \in e^\bullet \setminus {}^\bullet e$; $\chi'(b) = 1$ for all $b \in e^\bullet$; $\chi'(b) = 0$ for all $b \in {}^\bullet e \setminus e^\bullet$; and $\chi'(b) = \chi(b)$ for all $b \notin {}^\bullet e \cup e^\bullet$. As a consequence the configuration system of a 1-safe Petri net corresponds precisely to its usual marking graph.

The *language* $L(\mathcal{S})$ *of sequential computations* of $\mathcal{S}$ consists of all words $u = a_1 ... a_n \in \Sigma^\star$ for which there are some states $q_0, ..., q_n \in Q$ such that $\imath = q_0$ and for each $i \in [1, n]$, $q_{i-1} \xrightarrow{a_i} q_i$. For short, these conditions will be denoted by $q_0 \xrightarrow{u} q_n$. We can check that a configuration q is reachable if and only if $\imath \xrightarrow{u} q$ for some $u \in \Sigma^\star$. In the sequel of this section we consider implicitly only reachable configurations.

2.2 Modeling Concurrency with Independence Relations

Let us first recall some basic notions of Mazurkiewicz trace theory [7]. Let $\| \subseteq \Gamma \times \Gamma$ be a binary, symmetric and irreflexive relation over some alphabet Γ. The associated *trace equivalence* is the least congruence $\sim$ over $\Gamma^\star$ such that for all $a, b \in \Gamma$, $a\|b$ implies $ab \sim ba$. A *trace* $[u]$ is the equivalence class of a word $u \in \Gamma^\star$. We denote by $\mathbb{M}(\Gamma, \|)$ the set of all traces w.r.t. $(\Gamma, \|)$.

In [8] Droste introduced a generalization of Mazurkiewicz traces by providing each configuration with its own independence relation. We follow this approach verbatim in order to identify equivalent sequential computations.

Let $q = (\chi, s)$ be a configuration and $a \in \Sigma$ be an action such that $q \xrightarrow{a} q'$ for some q'. By Def. 1.4, there exists a single transition rule $\rho \in \Delta$ such that $a_\rho = a$, $\nu_\rho = \chi|\mathrm{R}_\rho$ and $s_\rho = s|a$. This particular transition rule is denoted by $\rho_{q,a}$. Note here

that the configuration system $\mathcal{C}(\mathcal{S})$ is *deterministic*: If $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$ then $q' = q''$.

Definition 2.2. *Let $q \in Q$ be some configuration and $a, b \in \Sigma$ be two actions. We put $a \|_q b$ if there are $q', q'' \in Q$ such that $q \xrightarrow{a} q'$, $q \xrightarrow{b} q''$ and $\rho_{q,a} \| \rho_{q,b}$.*

Thus two distinct actions are independent from each other in some configuration if they correspond to transition rules that may occur concurrently.

Now the independence relations $\|_q$ yield a natural equivalence relation over the set of sequential computations $L(\mathcal{S})$ as follows. The *trace equivalence* $\sim_{\mathcal{S}}$ is the least equivalence over $L(\mathcal{S})$ such that for all words $u, v \in \Sigma^\star$ and all actions $a, b \in \Sigma$ if $\imath \xrightarrow{u} p \xrightarrow{ab} q \xrightarrow{v} r$ and $a \|_p b$ then $u.ab.v \sim_{\mathcal{S}} u.ba.v$. If w and w' are two trace equivalent words then they lead from the initial configuration to the same configuration. For any word $u \in L(\mathcal{S})$, the *trace* $[u]$ consists of all words $v \in L(\mathcal{S})$ that are trace equivalent to u: Formally we put $[u] = \{v \in \Sigma^\star \mid v \sim_{\mathcal{S}} u\}$. The *trace language* $\mathcal{L}_t(\mathcal{S}) = L(\mathcal{S})/\sim_{\mathcal{S}}$ consists of all traces.

2.3 From Traces to Pomsets... and Back

Consider now again the set of all Mazurkiewicz traces $\mathbb{M}(\Gamma, \|)$. Let $u \in \Gamma^\star$; then the trace $[u]$ is precisely the set of linear extensions $\mathrm{LE}(t)$ of a unique pomset $t = (E, \preccurlyeq, \xi)$, that is, $[u] = \mathrm{LE}(t)$. Moreover t satisfies the following additional properties:

M_1: For all events $e_1, e_2 \in E$ with $\xi(e_1) \not\| \xi(e_2)$, we have $e_1 \preccurlyeq e_2$ or $e_2 \preccurlyeq e_1$;
M_2: For all events $e_1, e_2 \in E$ with $e_1 \multimap e_2$, we have $\xi(e_1) \not\| \xi(e_2)$.

Conversely the linear extensions of a pomset satisfying these two axioms form a trace of $\mathbb{M}(\Gamma, \|)$. Thus one usually identifies $\mathbb{M}(\Gamma, \|)$ with the class of pomsets satisfying M_1 and M_2.

Recall now that each transition $q \xrightarrow{a} q'$ corresponds to some transition rule $\rho_{q,a}$, so each computation sequence $u \in L(\mathcal{S})$ corresponds to a sequence of transition rules $\rho_u \in \Delta^\star$. Moreover two computation sequences u and v are trace equivalent w.r.t. $\sim_{\mathcal{S}}$ if and only if the corresponding words ρ_u and ρ_v are trace equivalent w.r.t. the May-Occur-Concurrently relation. It follows that the equivalence class $[u]$ is the set of linear extensions of some pomset t which corresponds to the Mazurkiewicz trace $[\rho_u]$. The next result shows that this pomset is accepted by $\mathcal{S}$. Moreover any pomset from $\mathcal{L}(\mathcal{S})$ corresponds to some trace of $\mathcal{L}_t(\mathcal{S})$.

Theorem 2.3. *For each $u \in L(\mathcal{S})$ we have $[u] = \mathrm{LE}(t)$ for some $t \in \mathcal{L}(\mathcal{S})$. Conversely for each $t \in \mathcal{L}(\mathcal{S})$ we have $[u] = \mathrm{LE}(t)$ for some $u \in L(\mathcal{S})$.*

The following result presents an efficient way to compute the pomset associated to some sequential computation inductively over the length of that computation.

Corollary 2.4. *Let $\imath = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ be a finite sequence of transitions in $\mathcal{C}(\mathcal{S})$. Let $t = (E, \preccurlyeq, \xi)$ be a pomset such that $E = \{e_1, \ldots, e_n\}$, $\xi(e_i) = a_i$ for each $i \in [1..n]$, $e_1 \leqslant e_2 \leqslant \ldots \leqslant e_n$ is a linear extension of t, and $\rho : e_i \mapsto \rho_{q_{i-1}, a_i}$ is a valid run of t. Then $\mathrm{LE}(t) = [a_1 \ldots a_n]$, ρ is a compatible run of t and $q_{\rho,E} = q_n$.*

3 Expressive Power of Deterministic Shared-Memory Systems

In this section we aim at characterizing the class of pomset languages that arise from shared-memory systems. We introduce first the notions of consistency and coherence in order to identify the expressive power of shared-memory systems (Theorem 3.7). Next we recall the definition of a regular set of pomsets from [11] and then we characterize the languages recognized by *finite* shared-memory systems (Theorem 3.14).

3.1 Consistency and Coherence

We borrow first the notion of a consistent set of pomsets from [2].

Definition 3.1. *A set of pomsets $\mathcal{L}$ is called* consistent *if*
$$\forall t_1, t_2 \in \mathrm{Pref}(\mathcal{L}) : \mathrm{LE}(t_1) \cap \mathrm{LE}(t_2) \neq \emptyset \Rightarrow t_1 = t_2.$$

In [2] this notion of consistency is restricted to prefix-closed sets of pomsets but we adopt here this relaxed definition in order to be able to extend this study to shared-memory systems provided with a set of final configurations in Section 4. Observe here that if $\mathcal{L}$ is a consistent set of pomsets and $\mathcal{L}' \subseteq \mathcal{L}$ then $\mathcal{L}'$ is consistent, too. Moreover $\mathcal{L}$ is consistent if and only if $\mathrm{Pref}(\mathcal{L})$ is consistent, too. Note also that Theorem 2.3 shows that the pomset language of any shared-memory system is consistent.

Example 3.2. Consider the two pomsets t_1 and t_2 from Fig. 3. The language $\mathcal{L} = \mathrm{Pref}\{t_1, t_2\}$ is not the pomset language of any SMS. Intuitively after the occurrence of events a and b some event c may occur in two different ways.

Let $(D, \leqslant)$ be a partial order. Two elements $d, d' \in D$ are compatible if they admit an upper bound. A subset $C \subseteq D$ is pairwise-compatible if any pair of elements of C admits an upper bound. The partial order $(D, \leqslant)$ is called *coherent* if any finite pairwise-compatible subset C admits an upper-bound. Recall now that pomsets are partially ordered by the prefix relation $\leqslant$.

Definition 3.3. *A set of pomsets $\mathcal{L}$ over Σ is* coherent *if $(\mathcal{L}, \leqslant)$ is coherent.*

Consider now a *consistent* set of pomsets $\mathcal{L}$. The *pomset equivalence* $\sim_{\mathcal{L}}$ over $\mathrm{LE}(\mathcal{L})$ is such that $w \sim_{\mathcal{L}} w'$ iff $\{w, w'\} \subseteq \mathrm{LE}(t)$ for some $t \in \mathcal{L}$. Note that $\sim_{\mathcal{L}}$ is an equivalence relation over $\mathrm{LE}(\mathcal{L})$ because $\mathcal{L}$ is consistent. The next proposition asserts that Definition 3.3 coincides with the notion of coherence from [16].

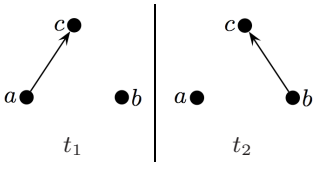


Fig. 3. A non-consistent set of pomsets

Proposition 3.4. *A prefix-closed and consistent set of pomsets $\mathcal{L}$ over Σ is coherent if and only if for all words $u \in \Sigma^\star$, all distinct actions $a, b, c \in \Sigma$:*

$$u.ab \sim_{\mathcal{L}} u.ba \wedge u.bc \sim_{\mathcal{L}} u.cb \wedge u.ca \sim_{\mathcal{L}} u.ac \text{ implies } u.abc \sim_{\mathcal{L}} u.acb \sim_{\mathcal{L}} u.cab.$$

For any SMS $\mathcal{S}$, Theorem 2.3 shows that $\mathcal{L}(\mathcal{S})$ is consistent, $L(\mathcal{S}) = \mathrm{LE}(\mathcal{L}(\mathcal{S}))$ and moreover $\sim_{\mathcal{S}}$ coincides with $\sim_{\mathcal{L}(\mathcal{S})}$. This enables us to check easily that the pomset language $\mathcal{L}(\mathcal{S})$ is coherent. Thus the pomset language of any shared-memory system is consistent, prefix-closed and coherent. Theorem 3.7 characterizes the expressive power of shared-memory systems by establishing the converse property.

3.2 Characterization of SMS Languages

Let Σ_1 and Σ_2 be two alphabets and $\pi : \Sigma_1 \to \Sigma_2$ a mapping from Σ_1 to Σ_2. This mapping extends into a map from $\Sigma_1^\star$ to $\Sigma_2^\star$. It extends also into a function that maps each pomset $t = (E, \preccurlyeq, \xi)$ over Σ_1 to the structure $\pi(t) = (E, \preccurlyeq, \pi \circ \xi)$. The latter might not be a pomset over Σ_2 in case some autoconcurrency appears in it. This situation can occur if $\pi(a) = \pi(b)$ for two distinct actions $a, b \in \Sigma$ while there are two events e and f in t that are labelled by a and b and that are not comparable. Refinements allow to relate two sets of pomsets $\mathcal{L}_1$ and $\mathcal{L}_2$ that are identical up to some relabeling.

Definition 3.5. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two* prefix-closed *sets of pomsets over Σ_1 and Σ_2 respectively. A mapping $\pi : \Sigma_1 \to \Sigma_2$ from Σ_1 to Σ_2 is a* refinement *from $\mathcal{L}_2$ onto $\mathcal{L}_1$ if $\pi(t)$ is a pomset for each $t \in \mathcal{L}_1$ and $\pi : \mathcal{L}_1 \to \mathcal{L}_2$ is a bijection.*

The main technical contribution of this section lies in the next lemma. A shared-memory system is called *singular* if the set of local states of each process $i \in \mathcal{I}$ is a singleton $S_i = \{\imath_i\}$. Furthermore a singular SMS is called *cellular* if $\Sigma = \mathcal{I}$ and $\mathrm{Loc}(a) = \{a\}$ for each $a \in \Sigma$. Asynchronous automata from [7], asynchronous cellular automata from [5], and 1-safe Petri nets are cellular shared-memory systems whereas generalized asynchronous cellular automata from [9] are singular shared-memory systems.

Lemma 3.6. *Let $\mathcal{L}$ and $\mathcal{L}'$ be two prefix-closed and consistent sets of pomsets over Σ and Σ' respectively such that there exists a refinement $\pi : \Sigma' \to \Sigma$ from $\mathcal{L}$ to $\mathcal{L}'$. If $\mathcal{L}'$ is recognized by a* cellular *SMS $\mathcal{S}'$ then $\mathcal{L}$ is recognized by a* singular *SMS $\mathcal{S}$ such that $\mathcal{S}$ and $\mathcal{S}'$ share the same configurations.*

We have explained above that the pomset language of any SMS is consistent, prefix-closed and coherent. The first result of this section establishes the converse property.

Theorem 3.7. *A set of pomsets is the language of some shared-memory system if and only if it is consistent, prefix-closed and coherent.*

Proof (sketch). The partial order of pomsets accepted by some SMS is isomorphic to the partial order traces of an asynchronous transition systems [4]. Therefore it corresponds to the configuration domain of a prime event structure with binary conflict. Thus it corresponds also to the marking graph of some occurrence net [19]. The Mazurkiewicz trace language of such an occurrence net is a refinement of $\mathcal{L}(\mathcal{S})$. By Lemma 3.6, $\mathcal{L}(\mathcal{S})$ is recognized by some singular SMS. ∎

3.3 Regular Sets of Pomsets

In the rest of this section we assume that the alphabet Σ is finite. We focus now on finite shared-memory systems, that is, we assume that any SMS consists of finitely many processes, local states, registers, and data. Thus a finite SMS admits finitely many configurations. Theorem 3.14 characterizes the class of pomset languages that arise from finite shared-memory systems by means of a notion of regularity borrowed from [11].

Let $t_1 = (E_1, \preccurlyeq_1, \xi_1)$ be a pomset over Σ. The *residual* $\mathcal{L} \setminus t_1$ consists of all pomsets $t_2 = (E_2, \preccurlyeq_2, \xi_2)$ such that there exists some pomset $t = (E, \preccurlyeq, \xi)$ in $\mathcal{L}$ satisfying the following conditions:

1. $E = E_1 \cup E_2$, $E_1 \cap E_2 = \emptyset$, and E_1 is an ideal of t,
2. t_1 is the restriction of t to events in E_1, and
3. t_2 is the restriction of t to events in E_2.

Definition 3.8. *Let $\mathcal{L}$ be a set of pomsets. Given two pomsets t and t', we put $t \equiv^r t'$ if $\mathcal{L} \setminus t = \mathcal{L} \setminus t'$. Then $\mathcal{L}$ is* regular *if the equivalence relation $\equiv^r$ is of finite index.*

Observe here that if $\mathcal{L}$ is a regular set of pomsets then $\mathrm{Pref}(\mathcal{L})$ is regular, too. Moreover Corollary 2.4 enables us to show that the pomset language of any finite SMS is regular.

Proposition 3.9. *For any finite SMS $\mathbb{S}$, the pomset language $\mathcal{L}(\mathbb{S})$ is regular.*

Consider now a *consistent* set of pomsets $\mathcal{L}$. For any two words $w, w' \in \Sigma^\star$, we put $w \equiv_{\mathcal{L}} w'$ if for all words u,$v \in \Sigma^\star$ it holds: $w.u \sim_{\mathcal{L}} w.v \Leftrightarrow w'.u \sim_{\mathcal{L}} w'.v$. It is easy to see that $\equiv_{\mathcal{L}}$ is a right-congruence over $\Sigma^\star$. The next lemma shows that Definition 3.8 corresponds to Arnold's notion of regularity [2] which was adopted in [16].

Lemma 3.10. *A consistent set of pomsets $\mathcal{L}$ is regular iff $\equiv_{\mathcal{L}}$ is of finite index.*

3.4 Two Powerful Ingredients

The characterization of the pomset languages that correspond to some finite SMS relies on two powerful ingredients, namely Zielonka's theorem [24] and some powerful but somewhat unrecognized result due to Arnold [2].

Definition 3.11. *A prefix-closed set of Mazurkiewicz traces $\mathcal{L} \subseteq \mathbb{M}(\Gamma, \|)$ is* forward-stable *w.r.t. $(\Gamma, \|)$ if for all words $u, v \in \Gamma^\star$ and all actions $a, b \in \Gamma$:*

$$[u.a] \in \mathcal{L} \wedge [u.b] \in \mathcal{L} \wedge a \| b \text{ implies } [u.ab] \in \mathcal{L}.$$

This condition is well-known. A forward-stable Mazurkiewicz trace language is called *safe-branching* in [22], *forward independence closed* in [18], *ideal* in [4], and *proper* in [15]. Let us now recall a particular version of Zielonka's theorem [18, 22, 24]: *Any forward-stable and prefix-closed regular set of Mazurkiewicz traces is accepted by a finite asynchronous automaton*. This result can be formulated in terms of refinement and 1-safe Petri nets as follows.

Theorem 3.12. *Let $\mathcal{L} \subseteq \mathbb{M}(\Gamma, \|)$ be a forward-stable and prefix-closed regular set of Mazurkiewicz traces. There exists a refinement from $\mathcal{L}$ to the language accepted by a finite 1-safe Petri net.*

Now the main contribution of [2] asserts that for any prefix-closed regular consistent set of pomsets $\mathcal{L}$ over Σ there exist a finite independence alphabet $(\Gamma, \|)$ and a refinement from $\mathcal{L}$ to a prefix-closed and regular set of Mazurkiewicz traces $\mathcal{L}' \subseteq \mathbb{M}(\Gamma, \|)$. By means of this strong result and Zielonka's theorem we proved in [16] the following statement.

Theorem 3.13. *[16, Cor. 4.6] Let $\mathcal{L}$ be a regular* coherent *prefix-closed consistent set of pomsets. There exists a refinement from $\mathcal{L}$ to a* forward-closed *and prefix-closed regular set of Mazurkiewicz traces $\mathcal{L}'$.*

By Theorem 3.7 and Proposition 3.9, the set of pomsets accepted by a finite SMS is regular, consistent, prefix-closed and coherent. Our main result depends again on the technical Lemma 3.6 and shows the converse property.

Theorem 3.14. *A set of pomsets is the language of a* finite *shared-memory system if and only if it is* regular, *consistent, prefix-closed, and coherent.*

Proof. Let $\mathcal{L}$ be a prefix-closed, consistent, regular and coherent set of pomsets. By Theorem 3.13, there exists a refinement π_1 from $\mathcal{L}$ to a forward-closed and prefix-closed regular set of Mazurkiewicz traces $\mathcal{L}_1$. By Theorem 3.12, there exists a refinement π_2 from $\mathcal{L}_1$ to the language $\mathcal{L}_2$ of a finite 1-safe Petri net. Then $\pi_2 \circ \pi_1 : \mathcal{L}_2 \to \mathcal{L}$ is a refinement from $\mathcal{L}$ to $\mathcal{L}_2$. By Lemma 3.6, $\mathcal{L}$ is the language of a finite singular SMS. ■

4 Comparisons with Related Works

In this section we provide shared-memory systems with a subset of final configurations. We derive from Theorem 3.14 two corollaries that are analoguous to some results from the theories of asynchronous automata and message-passing systems. Although we have considered Concurrent-Read-Exclusive-Write systems only, we explain also why our results apply to the setting of Exclusive-Read-Exclusive-Write systems, too.

4.1 Acceptance Condition and Deadlocks

We consider now finite shared-memory systems provided with an acceptance condition formalized by a subset of final configurations $F \subseteq Q$. So to say we have studied so far shared-memory systems for which all configurations are final. A pomset $t = (E, \preccurlyeq, \xi)$ is accepted by an SMS $\mathcal{S}$ with acceptance condition F, and we write $t \in \mathcal{L}_F(\mathcal{S})$, if there exists a linear extension $u \in \mathrm{LE}(t)$ such that $\mathrm{LE}(t) = [u]$ and moreover u leads from the initial configuration $\imath$ to some final configuration q within the configuration system $\mathcal{C}(\mathcal{S})$. Equivalently we require that the configuration $q_{\rho,E}$ belongs to F for some run ρ compatible with t. It is clear that the language $\mathcal{L}_F(\mathcal{S})$ of $\mathcal{S}$ is consistent. It is easy to check that $\mathcal{L}_F(\mathcal{S})$ is also regular.

Corollary 4.1. *A set of pomsets is the language of a finite shared-memory system with acceptance condition if and only if it is regular and consistent.*

Proof. Let $\mathcal{L}$ be a regular and consistent set of pomsets over Σ. We consider a new action $x \notin \Sigma$ and build the set of pomsets $\mathcal{L}^x$ by adding to any pomset from $\mathcal{L}$ a greatest event labeled by x. It is easy to see that $\mathcal{L}^x$ is regular and consistent. Furthermore

$\mathrm{Pref}(\mathcal{L}^x)$ is regular and consistent, too. By [2, Th. 6.16], there exists a refinement π from $\mathrm{Pref}(\mathcal{L}^x)$ onto a prefix-closed regular set of Mazurkiewicz traces $\mathcal{L}_1^x$ over some independence alphabet $(\Gamma, \|)$ (see also [16, Th. 3.5]). By [5], $\mathcal{L}_1^x$ is the language of a finite deterministic asynchronous cellular automaton with acceptance condition. Similarly to Lemma 3.6, we claim that $\mathrm{Pref}(\mathcal{L}^x)$ is the language of a finite singular SMS $\mathcal{S}^x$ with acceptance condition. We consider now the SMS $\mathcal{S}$ obtained from $\mathcal{S}^x$ as follows: First we forbid any occurrence of action x and second we consider any configuration q to be final if x is enabled from q in $\mathcal{S}^x$ and leads to a final configuration of $\mathcal{S}^x$. Then $\mathcal{S}$ is a singular SMS that accepts $\mathcal{L}$. ■

When dealing with a shared-memory system with acceptance condition the question arises whether it exhibits some deadlock, that is, a reachable configuration from which no final configuration is reachable. An SMS is deadlock-free if it admits no deadlock. Considering again the statement of Cor. 4.1, another interesting issue is to characterize which regular and consistent sets of pomsets are the language of some finite deadlock-free SMS. Let $\mathcal{S}$ be a finite shared-memory system with acceptance condition $F \subseteq Q$. Let $\mathcal{S}'$ be the SMS obtained from $\mathcal{S}$ by considering that all configurations are final. If $\mathcal{S}$ is deadlock-free then $\mathcal{L}(\mathcal{S}') = \mathrm{Pref}(\mathcal{L}_F(\mathcal{S}))$ hence $\mathrm{Pref}(\mathcal{L}_F(\mathcal{S}))$ is coherent (Theorem 3.7). Conversely, the next result shows that a regular and consistent set of pomsets $\mathcal{L}$ is recognized by some finite deadlock-free SMS as soon as $\mathrm{Pref}(\mathcal{L})$ is coherent.

Corollary 4.2. *A set of pomsets $\mathcal{L}$ is the language of some finite* deadlock-free *shared-memory system with acceptance condition if and only if $\mathcal{L}$ is regular and consistent* and moreover $\mathrm{Pref}(\mathcal{L})$ is coherent.

Proof. Let $\mathcal{L}$ be a regular consistent set of pomsets over Σ such that $\mathrm{Pref}(\mathcal{L})$ is coherent. We consider again the set of pomsets $\mathcal{L}^x$ obtained by adding to any pomset from $\mathcal{L}$ a greatest event labeled by a fixed new action $x \notin \Sigma$. As already observed, $\mathcal{L}^x$ is regular and consistent. It follows that $\mathrm{Pref}(\mathcal{L}^x)$ is regular and consistent, too. It is easy to check that $\mathrm{Pref}(\mathcal{L}^x)$ is coherent because $\mathrm{Pref}(\mathcal{L})$ is coherent. By Theorem 3.14, $\mathrm{Pref}(\mathcal{L}^x)$ is accepted by some finite SMS $\mathcal{S}^x$. We consider now the SMS $\mathcal{S}$ with acceptance condition obtained from $\mathcal{S}^x$ as follows: We forbid any occurrence of action x and we consider any configuration q to be final if x is enabled from q in $\mathcal{S}^x$. Then $\mathcal{S}$ accepts $\mathcal{L}$ and moreover $\mathcal{S}$ is deadlock-free. ■

These two corollaries can be regarded as an extension of Zielonka's theorem [5, 18, 22, 24] from the framework of deterministic asynchronous (cellular) automata to the setting of shared-memory systems. Both results can be extended to possibly infinite shared-memory systems with acceptance condition by dropping the regular condition. Note here also that Theorem 3.14 follows directly from Corollary 4.2.

4.2 Relationships with Communicating Finite-State Machines

Although this study does not aim at considering message-passing systems, this model fits somehow into the formalism of shared-memory systems provided that we consider only message-passing systems with bounded channels. The concurrent executions of message-passing systems are described by partial orders called message-sequence

charts (MSCs) [12]. When the system is regular then the number of messages in transit within channels is bounded and one can count messages within a channel modulo that bound. As observed formally by Kuske [13], these counters allow us to consider that messages are sent in different channels so that each channel contains at most one message at any stage of any execution. In that way it is possible to simulate any regular deterministic message-passing system by a finite shared-memory system.

At some more abstract level, any set of MSCs is a consistent set of pomsets. Moreover the usual notion of regularity for MSC languages [12] corresponds to the notion of regularity of sets of pomsets (Def. 3.8). Consequently Corollary 4.1 shows that any regular set of MSCs can be regarded as the language accepted by some finite SMS with acceptance condition.

Note that Corollaries 4.1 and 4.2 are analogous to some results from the theory of regular MSC languages. Namely, any regular set of MSCs is accepted by some finite-state deterministic message-passing system with bounded channels [12] whereas a characterization of the regular sets of MSCs that are accepted by some deadlock-free deterministic message-passing system is presented in [3] by means of a notion of coherence.

4.3 Exclusive-Read-Exclusive-Write Shared-Memory Systems

In this paper we have considered Concurrent-Read-Exclusive-Write shared-memory systems only. An alternative approach would be to consider *Exclusive-Read-Exclusive-Write* systems. In that case two distinct actions cannot read the value of the same register concurrently. This requires to add the next requirement in the definition of the May-Occur-Concurrently relation $\rho \| \rho'$: $\mathrm{R}_\rho \cap \mathrm{R}_{\rho'} = \emptyset$. With no surprise the behaviours of an EREW SMS can be represented by a regular, consistent, prefix-closed and coherent set of pomsets, too. Similarly to [7] we observe that any CREW SMS can be translated into some EREW SMS with the same number of reachable configurations and which accepts the same pomset language. Consequently all results from Sections 3 and 4 hold also in the setting of EREW shared-memory systems.

5 Conclusion

In this paper we have studied a partial-order semantics of shared-memory systems. This study has led to a concrete interpretation of consistent sets of pomsets (Theorem 3.7). Moreover we have identified the expressive power of finite shared-memory systems with the notion of regular consistent sets of pomsets (Theorem 3.14). Noteworthy our proofs rely on two major results by Arnold and Zielonka. Moreover this work depends heavily on our restriction to *deterministic* shared-memory systems.

In [17] we investigate the particular case of *unambiguous* shared-memory systems. An SMS is *unambiguous* if each pomset admits at most one compatible run. It is clear any deterministic SMS is unambiguous. However the pomset language recognized by some unambiguous SMS need not to be consistent. For instance the language $\mathrm{Pref}\{t_1, t_2\}$ from Example 3.2 is recognized by some unambiguous SMS. In [17] we establish a generalization of Arnold's result for non-consistent sets of pomsets. This

allows us to prove that a set of pomset is recognized by some unambiguous SMS if and only if it is definable in monadic second-order logic and satisfies a new property called media-boundedness.

At present we are investigating non-deterministic shared-memory systems and aiming at results analoguous to Corollaries 4.1 and 4.2 in that setting. Yet it is not clear so far whether any regular set of pomsets is recognized by some non-deterministic SMS.

References

1. Alur, R., Grosu, R.: Shared Variables Interaction Diagrams. In: 16th IEEE Int. Conf. on Automated Software Engineering, pp. 281–288. IEEE Computer Society, Los Alamitos (2001)
2. Arnold, A.: An extension of the notion of traces and asynchronous automata. RAIRO, Theoretical Informatics and Applications 25, 355–393 (1991)
3. 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)
4. Bednarczyk, M.: Categories of Asynchronous Systems. Ph.D. (Univ. of Sussex, 1988)
5. Cori, R., Métivier, Y., Zielonka, W.: Asynchronous mappings and asynchronous cellular automata. I&C 106, 159–202 (1993)
6. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific, Singapore (1995)
7. Diekert, V., Métivier, Y.: Partial Commutation and Traces. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, pp. 457–534 (1997)
8. Droste, M.: Concurrency, automata and domains. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 195–208. Springer, Heidelberg (1990)
9. Droste, M., Gastin, P., Kuske, D.: Asynchronous cellular automata for pomsets. TCS 247, 1–38 (2000)
10. Duboc, C.: Mixed product and asynchronous automata. TCS 48, 183–199 (1986)
11. Fanchon, J., Morin, R.: Regular Sets of Pomsets with Autoconcurrency. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 402–417. Springer, Heidelberg (2002)
12. Henriksen, J.G., Mukund, M., Narayan Kumar, K., Sohoni, M., Thiagarajan, P.S.: A Theory of Regular MSC Languages. I&C 202, 1–38 (2005)
13. Kuske, D.: Regular sets of infinite message sequence charts. I&C 187, 80–109 (2003)
14. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21, 558–565 (1978)
15. Mazurkiewicz, A.: Trace theory. LNCS, vol. 255, pp. 279–324. Springer, Heiedelberg (1987)
16. Morin, R.: Concurrent Automata vs. Asynchronous Systems. In: Jedrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 686–698. Springer, Heidelberg (2005)
17. Morin R.: Logic for Unambiguous Shared-Memory Systems. In: DLT 2008. LNCS. Springer, Heidelberg (accepted, 2008)
18. Mukund, M.: From global specifications to distributed implementations. In: Synthesis and Control of Discrete Event Systems, pp. 19–34. Kluwer, Dordrecht (2002)
19. Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, events structures and domains, part 1. TCS 13, 85–108 (1981)
20. Pratt, V.: Modelling concurrency with partial orders. International Journal of Parallel Programming 15, 33–71 (1986)
21. Sassone, V., Nielsen, M., Winskel, G.: Deterministic Behavioural Models for Concurrency (Extended Abstract). In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 682–692. Springer, Heidelberg (1993)

22. Ştefănescu, A., Esparza, J., Muscholl, A.: Synthesis of distributed algorithms using asynchronous automata. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 20–34. Springer, Heidelberg (2003)
23. Szpilrajn, E.: Sur l'extension de l'ordre partiel. Fund. Math. 16, 386–389 (1930)
24. Zielonka, W.: Notes on finite asynchronous automata. RAIRO, Theoretical Informatics and Applications 21, 99–135 (1987)
25. Zielonka, W.: Safe executions of recognizable trace languages by asynchronous automata. In: Meyer, A.R., Taitslin, M.A. (eds.) Logic at Botik 1989. LNCS, vol. 363, pp. 278–289. Springer, Heidelberg (1989)

A Scalable and Oblivious Atomicity Assertion

Rachid Guerraoui and Marko Vukolić*

School of Computer and Communication Sciences, EPFL,
INR, Station 14, CH-1015 Lausanne, Switzerland
{rachid.guerraoui,marko.vukolic}@epfl.ch

Abstract. This paper presents SOAR: the first oblivious atomicity assertion with polynomial complexity. SOAR enables to check atomicity of a single-writer multi-reader register implementation. The basic idea underlying the low overhead induced by SOAR lies in greedily checking, in a backward manner, specific points of an execution where register operations could be linearized, rather than exploring all possible precedence relations among these.

We illustrate the use of SOAR by implementing it in +CAL. The performance of the resulting automatic verification outperforms comparable approaches by more than an order of magnitude already in executions with only 6 read/write operations. This difference increases to 3-4 orders of magnitude in the "negative" scenario, i.e., when checking some non-atomic execution, with only 5 operations. For example, checking atomicity of every possible execution of a single-writer single-reader (SWSR) register with at most 2 write and 3 read operations with the state of the art oblivious assertion takes more than 58 hours to complete, whereas SOAR takes just 9 seconds.

1 Introduction

With multi-core architectures becoming mainstream, concurrent programming is expected to become the norm, even among average developers who might not always have the right skills and experience. Concurrent programming is however notoriously difficult. In particular, it is hard to control the interference between concurrent threads without compromising correctness on the one hand, or restricting parallelism on the other hand.

Among consistency criteria for concurrent programming, *atomicity* (also known as *linearizability* [15]) is one of the most popular. This is because atomicity reduces the difficult problem of reasoning about a concurrent program into the simpler problem of reasoning about its sequential counterpart. Roughly speaking, atomicity guarantees that concurrently-executing requests on shared objects appear sequential: namely, each request appears to be executed at some point (known as the *linearization point* [15]) between its invocation and response time (real-time ordering). An example of an atomic execution of a read/write register is depicted

* Current address: IBM Zurich Research Laboratory, Säumerstrasse 4, CH-8803, Rüschlikon, Switzerland.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 52–66, 2008.

in Figure 1, along with its linearization points (assuming the register is initialized to 0). In contrast, the execution in Figure 2 is not atomic. This is because we cannot place linearization points such that the sequential specification of a register is satisfied, i.e., every read returns the last value written.

Precisely because it simplifies the job of the programmers by encapsulating the difficulty underlying synchronizing shared atomic objects, atomicity is hard to implement. As pointed out in [7], an evidence of this difficulty is that several published implementations of atomic shared memory objects have later shown to be incorrect. Not surprisingly, tools for checking atomicity are of crucial importance, in particular automatic ones that are suitable for machine verification [11].

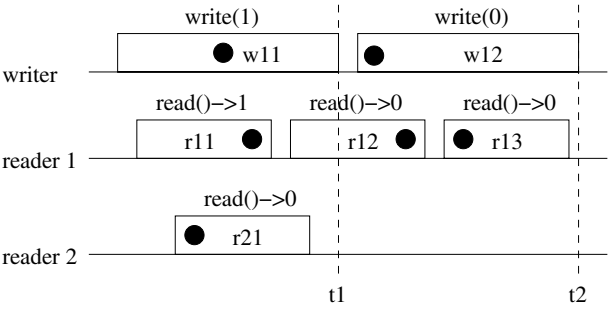


Fig. 1. Example of an atomic execution

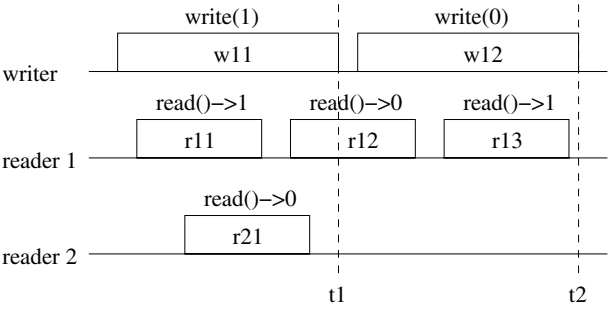


Fig. 2. Example of a non-atomic execution

So far, tools for checking atomicity have mainly been designed for specific programming languages (e.g., Concurrent Java [10]). Some exceptions have been proposed in the form of *language-oblivious* execution assertions, which enable to check the atomicity of implementation histories. Some of these (e.g., [16, 17]) are still *non-algorithm-oblivious* in the sense that a fair amount of knowledge about the checked algorithm is needed in order to check correctness.

Genuinely oblivious assertions were proposed in [25] (Lemma 13.16) and [19]. These assertions do not require any knowledge, neither about the language nor

about the checked algorithm. One specific such assertion is of particular interest: the one of Chockler et al. (Property 1 of [7]) for it was especially devised for automatic verification. This assertion, which we refer to as CLMT, can be written as a simple logical predicate and is very appealing for automatic verification especially when paired with a model checker such as the TLC for the +CAL algorithm language [23, 24].

Unfortunately, the CLMT assertion does not scale well as we discuss below. Consider the (non-atomic) execution on a single-writer multi-reader (SWMR) read/write register depicted in Figure 2. When implemented in +CAL, the CLMT assertion takes more than one minute on our 4 dual-core Opteron machine to verify that this execution is not-atomic. This is even without taking into the account the operations invoked by *reader*2. When considering a single operation of *reader*2, the verification takes hours.

On the other hand, it is very simple for a human to verify manually that the execution of Figure 2 is not-atomic. For the execution to be atomic, the linearization point of the write operation $w1$ must come before that of read $r11$, since $r11$ does not return the initial value 0. Similarly, $w2$ must be linearized before $r12$. This leaves $r13$ which violates the sequential specification of the read/write register, meaning that the execution is not atomic.

What makes CLMT slow is the very fact that it reasons about atomicity by identifying the adequate properties of a *precedence relation* among read/write operations. Namely, CLMT checks atomicity by establishing the existence of a precedence relation among operations that: a) is a non-reflexive partial order, and b) satisfies certain (five different) properties. Without diving into the details of these properties, it is easy to see that this verification scheme cannot scale for it does imposes an *exponential* computational complexity on a model checker. Namely, with $2^{|op|^2}$ different possible relations over the set of $|op|$ different operations, there is simply too many relations to check, even for modest values of $|op|$, regardless of the nature of the properties that are to be checked. This is especially true when the "good" precedence relation does not exist, i.e., when the execution is not atomic. The motivation of this paper is to ask whether it is possible to devise an oblivious, yet scalable atomicity assertion.

We present *SOAR (Scalable and Oblivious Atomicity asseRtion)*, the first oblivious atomicity assertion with polynomial complexity. SOAR is devised for single-writer multi-reader concurrent objects, of which the single-writer multi-reader register is a very popular representative [6, 25]. Indeed, many applications of the register abstraction make use mainly of its single-writer variant. Such applications include for example consensus [2, 5, 12] as well as snapshot implementations [3].

Like CLMT, SOAR gives a sufficient condition for atomicity; in fact, SOAR is equivalent to CLMT in our single-writer setting.[1] Interestingly, we could also use SOAR in +CAL to verify that some seemingly natural simplifications of the celebrated Tromp's algorithm [27] (implementing an atomic bit out of three safe bits) lead to incorrect solutions. By doing this, we show that our SOAR

[1] For lack of space, we omit the equivalence proof; it can be found in [14].

implementation in +CAL can be used successfully in identifying non-atomic executions and algorithm debugging.

SOAR has a low-degree polynomial complexity ($O(|op|^3)$ in the worst case). It outperforms CLMT [7] by more than an order of magnitude already in verifying atomicity of executions with only 6 read/write operations.[2] This difference increases to 3-4 orders of magnitude in the "negative" scenario, i.e., when checking some non-atomic execution. For example, checking atomicity of every possible execution of a single-writer single-reader (SWSR) register with at most 2 write and 3 read operations with CLMT takes more than 58 hours to complete, whereas SOAR takes just 9 seconds. As we pointed out however, SOAR is designed specifically for verifying atomicity of single writer objects, whereas CLMT is a general assertion suitable also for multi-writer applications.

Underlying SOAR lies the idea of *greedy linearization*. Basically, SOAR looks for linearization points in an execution ex rather than checks for precedence relations. SOAR performs its search in a backward manner starting from the end of the execution, linearizing the last write operation in ex (say w) and then trying to linearize as many read operations as possible *after* w. Then, the linearized operations are removed from ex and the linearization reiterates. It is important to emphasize that the greedy linearization is without loss of generality.

While SOAR is specified with an atomic read/write data structure in mind, we believe that it is not difficult to extend it to cover other atomic objects in which only one process can change the state of the object (single-writer). Extending SOAR and the underlying greedy linearization idea to optimize model checking of multi-writer objects is very interesting open problem. This is left as future work.

The rest of the paper is organized as follows. After giving some preliminary definitions in Section 2, we describe our assertion in details in Section 3. In Section 4 we illustrate how SOAR can be used for model checking Tromp's algorithm and its variations in +CAL/TLC. We also report on some performance measurements. We conclude the paper with the related work overview in Section 5.

2 Preliminaries

2.1 Processes and Objects

We model processes and shared objects using the non-deterministic I/O Automata model [26]. We simply give here the elements that are needed to recall atomicity, state our assertion and prove its correctness. In short, an I/O automaton is a state machine whose state can change by discrete atomic transitions called *actions*. We consider two sets of processes: a singleton *writer* and a set of processes called *readers* (we refer to a process belonging to the union of these sets as *client*).

[2] We always compare SOAR to a version of CLMT that is optimized for the single-writer case as we discuss in Section 4.2.

A read/write register is a shared object consisting of the following:

1. set of values D, with a special value v_0 (called the initial value),
2. set of operations $write(v)$, $v \in D$ and $read()$
3. set of responses $D \cup \{ack\}$,
4. sequential specification of the register is any sequence of read/write operations such that the responses of operations comply with the following:
   (a) $write(v) \triangleq x := v$; return ack (where x is initialized to v_0)
   (b) $read() \triangleq$ return x

To access the register, a client issues an *operation descriptor* that consists of the identifier of the operation $id \in \{`write', `read'\}$ and the identifier of the client; in case of a write, a value v is added to the descriptor. To simplify the presentation, we sometimes refer to an operation descriptor op simply as an operation op. A single-writer multi-reader (SWMR) register is a read/write object in which only the process *writer* may issue write operations. We denote by wrs (resp., rds) the set of write (resp., read) operations.

Clients use the actions of the form $\mathsf{invoke}(op)$ and $\mathsf{response}(op, v)$, where $op \in wrs \cup rds$ and $v \in D \cup \{ack\}$, to invoke operations and to receive responses. A sequence β of invoke and response actions is called an *execution*. An invoked operation op is said to be complete (in some execution β) if β contains $\mathsf{response}(op, v)$, for some $v \in D \cup \{ack\}$ (we say $\mathsf{response}(op, v)$ *matches* $\mathsf{invoke}(op)$). An operation op is said to be *pending* in β if β contains the $\mathsf{invoke}(op)$ action but not its matching response.

The execution ex is *sequential* if (a) the first action is an invocation, (b) each invocation, except possibly the last, is immediately followed by its matching response, and (c) every response is immediately followed by an invocation.

We say that an execution β is *well-formed* if (1) for every $\mathsf{response}(op, v)$ action in β there is a unique $\mathsf{invoke}(op)$ action in β that precedes $\mathsf{response}(op, v)$, (2) for every client c there is at most one pending operation issued by c in β.

Moreover, we assume that each well-formed execution β contains the invocation and the response action of the special operation $w_0 = write(v_0)$ called the *initial write*, such that the response action for w_0 precedes invocations of any other operation. All executions considered in this paper are assumed to be well-formed. A well-formed, sequential execution β is called *legal*, if β is in the sequential specification of the register.

Finally, we say that a complete operation op *precedes* an operation op' (or, alternatively, that op' *follows* op) in a well formed execution β if the response action of op precedes the invocation action of op' in β (we denote this by $op <_\beta op'$). Let op and op' be two invoked operations in β; if neither $op <_\beta op'$), nor $op' <_\beta op$), we say that op and op' are *concurrent* (in β).

2.2 Atomicity

We define atomicity (or linearizability) in the following way [6]: a (well-formed) execution β is atomic if there is a permutation $\pi(\beta)$ of all operations in ex such that: (1) $\pi(ex)$ is legal, and (2) if $op <_\beta op'$ then $op <_{\pi(\beta)} op'$.

In this paper we rely on the *Partial Order (PO)* property [7] for proving atomicity. As shown in Lemma 2 of [7], PO is sufficient for atomicity, i.e., if β satisfies PO then β is atomic.

Definition 1 (*PO Property*). *Let op be the set of all operations invoked in the execution β that contains no pending operations and wrs (resp., rds) subset of all writes (resp., reads) in op. An execution β satisfies a* Partial Order *(PO) property if there is an irreflexive partial ordering $\prec$ on all elements of op, such that, in β:*

1. $\forall \pi, \phi \in op$, *if* $\pi <_\beta \phi$, *then* $\neg(\phi \prec \pi)$.
2. $\forall \pi, \phi \in wrs$, *either* $\pi \prec \phi$ *or* $\phi \prec \pi$.[3]
3. $\forall \pi \in wrs, \forall \phi \in rds$, *if* $\pi <_\beta \phi$, *then* $\pi \prec \phi$.
4. $\forall \pi, \phi \in rds$, *if* $\pi <_\beta \phi$ *then for each* $w \in LastPrecWrites(\pi, \prec)$, $w \prec \phi$.
5. *Let* $\pi \in rds$ *and let* v *be the value returned by* π. *Then,* v *is written by some write* $w \in LastPrecWrites(\pi, \prec)$.

Above, $LastPrecWrites(\pi, \prec) == \{w \in wrs : (w \prec \pi) \wedge \neg(\exists w' \in wrs : (w \prec w') \wedge (w' \prec \pi)\}$.

The PO property can be simply written as a logical predicate (assertion), to which we refer as CLMT.

3 A Scalable and Oblivious Atomicity asseRtion (SOAR)

3.1 Intuition: Greedy Linearization

Our SOAR assertion is motivated by the observation that it is easy to linearize (in the single-writer case) the fragments of the execution between every two writes. Consider for example the fragment of the execution of Figure 2 in between initial time t_0 and time t_1, the time of completion of write $w1$, that contains only those read operations that are invoked before t_1 (i.e., r_{11}, r_{12} and r_{21}). It is clear that only read operations that return the value written by w_0 (say v_0) can be linearized between w_0 and w_1. Moreover, such reads cannot be preceded by reads that return values other than v_0. In other words, in the execution of Figure 2, only r_{21} can be linearized between w_0 and w_1 while the other reads must be linearized after w_1. We can repeat this partitioning of the execution between two writes and apply the above reasoning iteratively, until we exhaust all write operations. When a single write operation w_W is left, the remaining (still non-linearized) read operations must return the value written by w_W in order for the execution to be atomic. In the example of Figure 2 the operations would be linearized in the following order: w_0, r_{21}, w_1, r_{11}, w_2, r_{12}, leaving r_{13} which actually violates the sequential specification of the atomic read/write register.

The greedy linearization idea described above is based on checking the fragments of the execution that are between every two writes, starting from the

[3] In our case, with the single writer, this property becomes (having in mind Property 1) becomes: $\forall \pi, \phi \in wrs$, if $\pi <_\beta \phi$ then $\pi \prec \phi$.

beginning of the execution. While this is the natural way for a human to linearize executions, this approach leads to reasoning about execution suffixes (that remain after removing linearized operations). In our case, we found it more convenient to reason formally about execution prefixes; hence, we choose to apply greedy linearization starting from the end of the execution, using the similar idea. Consider, again the execution of Figure 2. It is trivial to see that the last write to be linearized is w_2. Now we can try to linearize as many reads as possible *after* w_2; however, this cannot be done with any of the reads. We can remove all linearized operations from the execution (i.e., in our case, only w_2) and apply the same reasoning to the remaining execution prefix. However, before reiterating, we must make sure that removing linearized operations indeed leaves us with the execution prefix; more concretely, we must check that none of the reads that will remain in the execution was invoked after the completion of the linearized write. In the case of w_2, this condition is satisfied (no operations are invoked after w_2 completes). In the next iteration, we would linearize w_1 and r_{13}. Finally, in the last iteration we could see that the atomicity is violated since not all of the remaining read operations return the value written by the initial write (r_{11} returns 1).

3.2 Description

We formalize our greedy linearization approach to obtain a generic assertion for atomicity in the following way. We denote:

- by W the total number of writes (not counting the initial write) in some execution ex that contains no incomplete operations,
- by w_i the i^{th} write in ex,
- by rds_W the set of all read operations in ex, and
- by $ex_i^{rds_i}$ $(i = 0 \ldots W)$ the prefix of the execution ex that contains only write operations from w_0 to w_i, and only read operations from set rds_i.

Notice that $ex_W^{rds_W} \equiv ex$.

We assert the atomicity of every partial execution $ex_i^{rds_i}$ $(i = 0 \ldots W)$ as follows:

1. If $i = 0$ (i.e., if $ex' = ex_0^{rds_0}$ contains only one write) then ex' is atomic if and only if all (read) operations from rds_0 return the initial value,
2. else (i.e., if $i > 0$), we:
   (a) remove from rds_i every read r that satisfies the following properties (we denote the set of such reads $linRds(i)$):
      i. r returns the value written by the write w_i,
      ii. r does not precede w_i, and
      iii. if some $r' \in R_i$ follows r, then r' returns the value written by w_i.

   Basically, the reads from the set $linRds(i)$ are immediately linearizable and SOAR greedily linearizes such reads.
   (b) If there is a read in $rds_i \setminus linRds(i)$ that follows w_i, then $ex_i^{rds_i}$ is not atomic.

(c) $ex_i^{rds_i}$ is atomic if and only if $ex_{i-1}^{rds_{i-1}}$ is atomic, where $rds_{i-1} = rds_i \setminus linRds(i)$.

Given the recursive nature of SOAR, the assertion can be written more compactly (and more precisely) as a logical predicate (Figure 3). We write it as follows, using the TLA+ [22].

$$
\begin{array}{l}
linRds(wrs, rds, Inv, Resp, Ret) == \{r \in rds : \\
\quad \bigwedge\ Ret[r] = Ret[lastWR(wrs)] \\
\quad \bigwedge\ Resp[r] > Inv[lastWR(wrs)] \\
\quad \bigwedge\ \forall r' \in rds : Resp[r] < Inv[r'] \Rightarrow Ret[r'] = Ret[lastWR(wrs)]\} \\
\\
SOAR(wrs, rds, Inv, Resp, Ret) == \\
\quad \text{IF } wrs = \{lastWR(wrs)\} \\
\quad \text{THEN } \forall r \in rds : Ret[r] = Ret[lastWR(wrs)] \\
\quad \text{ELSE} \\
\qquad \bigwedge\ \forall r \in rds \setminus linRds(wrs, rds, Inv, Resp, Ret) : \neg(Inv[r] > Resp[lastWR(wrs)]) \\
\qquad \bigwedge SOAR(wrs \setminus \{lastWR(wrs)\}, rds \setminus linRds(wrs, rds, Inv, Resp, Ret), Inv, Resp, Ret)
\end{array}
$$

Fig. 3. SOAR as a TLA+ predicate

In Figure 3, $SOAR()$ takes five arguments: (i) the sets wrs and rds containing the identifiers of all write and read operations in the execution ex, respectively, (ii) the functions (arrays) $Inv, Resp : wrs \cup rds \rightarrow Nat$ (where Nat is the set of natural numbers), containing the global logical time [18] of invocations and responses of operations, respectively, and (iii) the function (array) $Ret : wrs \cup rds \rightarrow D$ (where D is the domain of values that an implemented read/write register can assume), which maps the operations to values which are written/read. Moreover, SOAR makes use of the function $lastWR(wrs)$ which returns the write in wrs that follows all other writes in ex.[4]

It is not difficult to see that the very approach that underlies SOAR yields a low degree polynomial complexity ($O(|op|^3)$ in the worst case, where op is the number of operations in the execution), which is to be contrasted with the exponential one of the CLMT assertion.

To establish the correctness of SOAR we rely on the CLMT assertion, defined by the PO property, Def. 1, Section 2.2. We prove the correctness of SOAR by showing its equivalence with CLMT (in our single writer multi-reader model), using the following Lemmas:

Lemma 1. *If the assertion SOAR of Figure 3 applied on the sequence of read/write operations β returns $TRUE$, then β satisfies the PO property.*

[4] For simplicity, we assume that the identifiers of write operations are monotonically increasing with the time of operation invocation. If this is not the case the $lastWR()$ function should also take the function $Inv()$ as the argument.

Lemma 2. *If the sequence of read/write operations β satisfies the PO property, then the assertion SOAR of Figure 3 applied on β returns TRUE.*

Due to lack of space, the proofs of Lemmas 1 and 2 are omitted; these can be found in [14].

4 Application to Tromp's Algorithm

We applied SOAR and CLMT to the celebrated algorithm of Tromp [27] which we implemented in the +CAL algorithm language. We compared SOAR and CLMT performance, and evaluated SOAR's applicability to detection of non-atomic executions and, hence, to debugging.

Our +CAL implementation of Tromp's algorithm with SOAR is given in Figure 4.[5] It consists of three parts: (1) The code used for testing (given in lines 109-120), (2) the SOAR part (comprised of lines 006-011, 037-043, 058-060, 064-068 and 102-106), and (3) the +CAL implementation of the Tromp's algorithm (comprised of the remaining lines of Figure 4). We explain both parts of the code, starting with Part 2 (Tromp's algorithm). In the following we refer to Figure 4.

In short, Tromp's algorithm gives an implementation of a single-writer single-reader (SWSR) atomic bit, using 3 safe[6] [19] (SWSR) bits: V, W and R, all initialized to 0. Bits V and W are owned (written) by the writer, whereas R is owned by the reader of the atomic bit. To simulate safe registers in +CAL, we use the variables *busy* and *value* (lines 012-014), as well as macros in lines 020-034. The main code of the Tromp's algorithm is given in lines 044-057 (the write code) and 069-101 (the read code). Comments in these portions of code (e.g., in lines 046, 050, or 070, 076, etc.) give the lines of the pseudocode as stated in the original paper [27]. Below each such comment, there is a +CAL translation of the corresponding pseudocode.

The SOAR part of the code in Figure 4 consists of operations on certain history variables necessary for the implementation of SOAR (as well as CLMT). History variables [1] play no role in the algorithm and serve only for the assertions. These lines are written as a wrapper around the code of the original algorithm in an *oblivious* manner; namely, no lines are inserted in the main code of Tromp's algorithm. For example, lines 037-043 and 058-060 are wrapped around the original write code, whereas lines 064-068 and 102-106 are wrapped around the original read code. Below, we explain in details the history variables required by SOAR.

First, SOAR requires history variables wrs and rds (sets of write and read operations), as well as history arrays (functions) $Inv[]$, $Resp[]$ and $Ret[]$. Initially, $wrs = \{0\}$, i.e., wrs contains the identifier of the initial write w_0, while $Inv[0] =$

[5] In Figure 4, '043:' denotes a line number added for simplicity of presentation, whereas 'l17:' denotes a +CAL label.

[6] Basically, a safe register ensures that a read rd returns the last value written only if rd is not concurrent with any write. In case of concurrency, a read may return an arbitrary value.

```
001: ------------------------------ MODULE TrompSOAR2 ------------------------------
002: EXTENDS Naturals, TLC, Sequences
003: CONSTANT MAXWRITE, MAXREAD, V, W, R, WRITER, READER, SOAR(_,_,_,_,_)

004: (* –algorithm Tromp
005:   variables
006:     (* 1. History variables used by SOAR and CLMT. *)
007:     globalClock = 0, writeCount = 0,
008:     readCount = MAXWRITE,wrs = {0},rds = {},
009:     Ret = [i ∈ 0 . . . MAXWRITE + MAXREAD ↦ 0],
010:     Inv = [i ∈ 0 . . . MAXWRITE + MAXREAD ↦ 0],
011:     Resp = [i ∈ 0 . . . MAXWRITE + MAXREAD ↦ 0]

012:     (* 2. Variables used to simulate safe registers. *)
013:     busy = [i ∈ {V, W, R} ↦ FALSE],
014:     value = [i ∈ {V, W, R} ↦ 0],

015:     (* 3. Tromp's algorithm variables. *)
016:     oldValue = 0,   (* Used by the writer*)
017:     R_writer = 0,   (* Used by the writer to read R*)
018:     W_reader = 0,   (* Used by the reader to read W*)
019:     v = 0, x = 0, returnValue = 0 (* Used by the reader*)

020:   (* 4. Safe register simulation. *)
021:   macro RW_INIT(reg) begin
022:     if ⋁((reg ∈ V, W) ∧ (self = WRITER))
023:        ⋁((reg = R) ∧ (self = READER))
024:     then busy[reg] := TRUE;
025:     end if;
026:   end macro

027:   macro READ(reg, result) begin
028:     if busy[reg] = FALSE then result := value[reg];
029:     else either result := 0 or result := 1 end either;
030:     end if;
031:   end macro

032:   macro WRITE(reg, val) begin
033:     value[reg] := val; busy[reg] := FALSE;
034:   end macro

035:   (* 5. Tromp's Algorithm w. SOAR. *)
036:   procedure write(val) begin l1:
037:     (* Update of history variables*)
038:     writeCount := writeCount + 1;
039:     globalClock := globalClock + 1;
040:     Inv[writeCount] := globalClock;
041:     Resp[writeCount] := INF;
042:     Ret[writeCount] := val;
043:     wrs := wrs ∪ {writeCount};

044:     (* Tromp's algorithm write() code*)
045:       if oldValue ≠ val then
046:         (* change V *)
047: l2:      RW_INIT(V);
048: l3:      WRITE(V, val);
049:         oldValue := val;
050:         (* if W=R then change W *)
051: l4:      RW_INIT(R);
052: l5:      READ(R, R_writer);
053:         if value[W] = R_writer then
054: l6:        RW_INIT(W);
055: l7:        WRITE(W, 1 − value[W]);
056:         end if;
057:       end if;

058: l8:    (* Update of history variables and return*)
059:       globalClock := globalClock + 1;
060:       Resp[writeCount] := globalClock;
061:       return;
062:   end procedure

063:   procedure read() begin l9:
064:     (* Update of history variables *)
065:     globalClock := globalClock + 1;
066:     readCount := readCount + 1;
067:     rds := rds ∪ {readCount};
068:     Inv[readCount] := globalClock;

069:     (* Tromp's algorithm read() code*)
070:       (* If W=R return v - line 1 *)
071: l10:   RW_INIT(W);
072: l11:   READ(W,W_reader);
073:       if W_reader = value[R] then
074:         returnValue := v;
075:       else
076:         (* x : = Read(V) - line 2 *)
077: l12:     RW_INIT(V);
078: l13:     READ(V, x);
079:         (* If W≠R change R - line 3 *)
080: l14:     RW_INIT(W);
081: l15:     READ(W, W_reader);
082:         if W_reader ≠ value[R] then
083: l16:       RW_INIT(R);
084: l17:       READ(R, 1 − value[R]);
085:         end if;
086:         (* v : = Read(V) - line 4 *)
087: l18:     RW_INIT(V);
088: l19:     READ(V, v);
089:         (* If W=R return v - line 5 *)
090: l20:     RW_INIT(W);
091: l21:     READ(W, W_Reader);
092:         if W_reader = value[R] then
093:           returnValue := v;
094:         else
095:           (* v : = Read(V) - line 6 *)
096: l22:       RW_INIT(V);
097: l23:       READ(V, v);
098:           (* return x - line 7 *)
099:           returnValue := x;
100:       end if;
101:     end if;

102: l24:(* Update of history variables and return*)
103:     Ret[readCount] := returnValue;
104:     globalClock := globalClock + 1;
105:     Resp[readCount] := globalClock;
106:     assert (SOAR(wrs, rds, Inv, Resp, Ret));
107:     return;
108:   end procedure

109:   (* 6. Code for testing. *)
110:   process Writer = WRITER begin wrloop:
111:     while (writeCount < MAXWRITE) ∧ (readCount ≤ MAXWRITE + MAXREAD) do
112:       either call write(0) or call write(1) end either;
113:     end while;
114:   end process;

115:   process Reader = READER begin rdloop:
116:     while (writeCount ≤ MAXWRITE) ∧ (readCount < MAXWRITE + MAXREAD) do
117:       call read();
118:     end while;
119:   end process;
120: end algorithm
*)
```

Fig. 4. SOAR application to Tromp's algorithm

$Resp[0] = 0$ and $Ret[0] = v_0 = 0$ (lines 008-011). Besides, the following three history variables are also needed (line 007): (i) *globalClock* to act as a global clock and, hence, facilitate the implementation of functions $Inv[]$ and $Resp[]$, and (ii) *writeCount* and *readCount* the counters for write and read operation identifiers, respectively, which take values from non-overlapping domains. All these variables/arrays are accessed only at the beginning (invocation) and the end (completion) of read/write operations. We believe that the operations on history variables are very intuitive and simple to follow. We clarify, however, two lines: (a) in line 041, the response time of the newly invoked write is set to INF, where INF (infinity) represents a constant that such that the *globalClock* cannot get greater than INF, and (b) in line 103, the returned value of the read is taken from the $returnValue$ variable in which the main read code of Tromp's algorithm (lines 069-101) stores the read value.

Finally, constants $MAXWRITE$ (resp., $MAXREAD$) denote the maximum number of write (resp., read) operations invoked in the checked execution.

4.1 Asserting Non-atomic Executions

We used our implementation of Figure 4 to verify that certain, seemingly plausible, "optimizations" of Tromp's algorithm lead to the incorrect solution.

For example, it is not straightforward to see why the condition 'if $W \neq R$' in line 3 of the Tromp's read pseudocode is necessary (see line 079, Fig. 4) knowing that this line is executed only if indeed $W \neq R$ in line 1 of the original read pseudocode (line 070, Fig. 4). However, removing this condition (i.e., lines 080-082 and 085 of Fig. 4) leads to a violation of atomicity, which can be detected by SOAR. Using the error output of the TLC model checker, we were able to extract

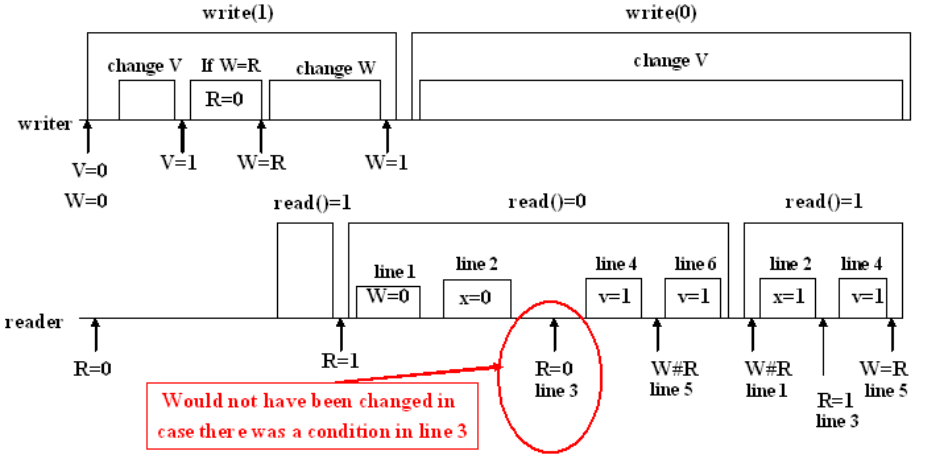


Fig. 5. Violation of atomicity after removing the condition in line 3 of Tromp's algorithm

the execution that leads to the atomicity violation (see Figure 5). Interestingly, such "simplified" Tromp's algorithm remains regular [19], but it is not atomic. In a similar way, we were also able to show that the instruction in line 6 of the original pseudocode (line 095, Fig. 4) is also necessary. This demonstrates the usability of SOAR in debugging and asserting non-atomicity in practice.

4.2 Performance

All our performance results are obtained running TLC model checker (using 4 processors) on a 4 dual-core Opteron 8216 with 8 GB of RAM. TLC model checker is ran on an implementation of the Tromp's algorithm in +CAL, varying the number of invoked read/write operations.

Model checking was done to verify the atomicity of the Tromp's algorithm using both the CLMT and SOAR. Obtained graphs are given in Figure 6. Results are given for a specific variation of the CLMT, optimized for a single writer scenario. Notably the optimization modifies the condition 2 of Definition 1, Section 2.2 to impose that for any precedence relation $\prec$ and every $i \in 1 \ldots W-1$ $w_i \prec w_{i+1}$ (where W is the total number of writes, represented by the variable $writeCount$ in our +CAL implementation, Figure 4). Moreover, the initial write w_0 was always pre-linearized before running the CLMT assertion, which significantly improves its performance.

From Figure 6 it can be seen that already in model checks of Tromp's algorithm with as few as 6 read/write operations (e.g., with 3 reads and 3 writes) a model check with SOAR takes more than an order of magnitude less time than with CLMT. The difference is even more glaring if a non-atomic execution is checked. For example, it takes only 15 milliseconds for SOAR to state that an

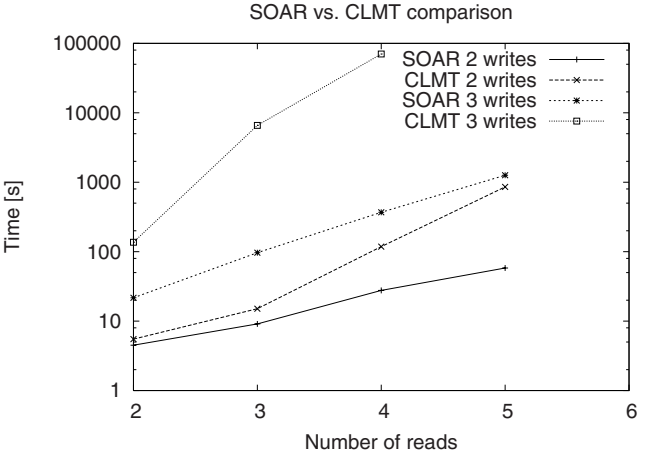


Fig. 6. SOAR vs. CLMT comparison. The entire time required for model checking Tromp's algorithm is represented.

execution of 2 (without the read r_{21}) is not atomic, whereas CLMT takes more than 70 seconds. This represents a difference of 3-4 orders of magnitude already for an execution with only 5 operations, and, by its design, the complexity of CLMT grows exponentially with the number of operations in the execution.

In practice, when checking executions with a fairly small number of operations, SOAR is as fast as any assertion maintaining the global clock can be. By maintaining the global clock, we mean maintaining the execution history in the form of: 1) set of all operations invoked in the execution, 2) arrays of operations' invocation and response times, and 3) the array of values written/read by operations. Indeed, our results show that, for all the points represented in Figure 6, SOAR introduces no visible overhead with respect to a dummy assertion that maintains the global clock.

5 Concluding Remarks

The concept of an *atomic* object was first introduced by Lamport [20, 21] in the context of read/write registers. This concept was later extended to objects other than registers by Herlihy and Wing [15], under the notion of *linearizability*. In this paper, we use notions of atomicity and linearizability interchangeably.

Atomicity assertions were proposed by Hesselink [16, 17]. These assertions are not *oblivious* since they are based on the history variables that are inserted in specific places of the checked algorithm. A fair amount of knowledge of the checked algorithm is thus required.

As we discussed in the introduction, Chockler et al., [7] proposed a genuinely *oblivious* atomicity assertion (quoted CLMT) that does not require any knowledge, neither on the language nor on the algorithm. In [7], CLMT has been used as the basis for the Partial Order machine automaton, that was in turn used in forward simulations to prove the correctness of various atomic object implementations (another simulation based atomicity proof (of a lock-free queue) can be found in the paper by Doherty et al. [8]). However, as we show in this paper, CLMT imposes exponential complexity on the model checker. This is not surprising given the result of Alur et al. [4], showing that model checking linearizability is in EXPSPACE. SOAR circumvents this result by focusing on the single-writer implementations.

In [27], Tromp proposed an atomicity automaton suitable for designing and verifying atomic variable constructions. The automaton nodes represent the state of a run on the atomic variable, whereas transitions represent read and write operations. This automaton addresses only the single-writer single-reader atomic constructions.

Some work was also devoted to checking the atomicity of transactional blocks of code, e.g., [9, 10, 13].

The simple greedy linearization idea that we employ in this paper is not new. A similar idea was exploited by Wang and Stoller [28] as one of the steps in the context of atomicity inference for programs with non-blocking synchronization.

Acknowledgments

We thank Gregory Chockler, Eli Gafni and Leslie Lamport for interesting discussions and very useful comments.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. 82(2), 253–284 (1991)
2. Abraham, I., Chockler, G.V., Keidar, I., Malkhi, D.: Byzantine disk paxos: optimal resilience with Byzantine shared memory. Distributed Computing 18(5), 387–408 (2006)
3. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM 40(4), 873–890 (1993)
4. Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. 160(1-2), 167–188 (2000)
5. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. Journal of Algorithms 11(3), 441–461 (1990)
6. Attiya, H., Welch, J.: Distributed Computing. Fundamentals, Simulations, and Advanced Topics. McGraw-Hill, New York (1998)
7. Chockler, G., Lynch, N., Mitra, S., Tauber, J.: Proving atomicity: An assertional approach. In: Proceedings of the 19th International Symposium on Distributed Computing, pp. 152–168 (September 2005)
8. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
9. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL 2004: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 256–267. ACM, New York (2004)
10. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI 2003: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 338–349. ACM, New York (2003)
11. Flanagan, C., Qadeer, S.: Atomicity for reliable concurrent software. In: A tutorial at the ACM SIGPLAN 2005 conference on Programming language design and implementation (PLDI 2005) (2005)
12. Gafni, E., Lamport, L.: Disk paxos. Distributed Computing 16(1), 1–20 (2003)
13. Guerraoui, R., Henzinger, T., Jobstmann, B., Singh, V.: Model checking transactional memories. In: PLDI 2008: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation (2008)
14. Guerraoui, R., Vukolić, M.: A scalable and oblivious atomicity assertion. Technical Report LPD-REPORT-2008-011, EPFL, School of Computer and Communication Sciences, Lausanne, Switzerland
15. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
16. Hesselink, W.H.: An assertional criterion for atomicity. Acta Informatica 38(5), 343–366 (2002)

17. Hesselink, W.H.: A criterion for atomicity revisited. Acta Informatica 44(2), 123–151 (2007)
18. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
19. Lamport, L.: On interprocess communication. Distributed computing 1(1), 77–101 (1986)
20. Lamport, L.: On interprocess communication. part i: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
21. Lamport, L.: On interprocess communication. part ii: Algorithms. Distributed Computing 1(2), 86–101 (1986)
22. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Reading (2002)
23. Lamport, L.: The +CAL algorithm language. In: NCA 2006: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications. IEEE Computer Society, Los Alamitos (2006)
24. Lamport, L.: Checking a multithreaded algorithm with +CAL. In: Proceedings of the 20th International Symposium on Distributed Computing, pp. 151–163 (September 2006)
25. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Mateo (1996)
26. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
27. Tromp, J.: How to construct an atomic variable (extended abstract). In: Proceedings of the 3rd International Workshop on Distributed Algorithms, London, UK, pp. 292–302. Springer, Heidelberg (1989)
28. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In: PPoPP 2005: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 61–71. ACM, New York (2005)

R-Automata*

Parosh Aziz Abdulla, Pavel Krcal, and Wang Yi

Department of Information Technology,
Uppsala University, Sweden
{parosh,pavelk,yi}@it.uu.se

Abstract. We introduce *R-automata* – finite state machines which operate on a finite number of unbounded counters. The values of the counters can be incremented, reset to zero, or left unchanged along the transitions. R-automata can be, for example, used to model systems with resources (modeled by the counters) which are consumed in small parts but which can be replenished at once. We define the language accepted by an R-automaton relative to a natural number D as the set of words allowing a run along which no counter value exceeds D. As the main result, we show decidability of the universality problem, i.e., the problem whether there is a number D such that the corresponding language is universal. We present a proof based on finite monoids and the factorization forest theorem. This theorem was applied for distance automata in [12] – a special case of R-automata with one counter which is never reset. As a second technical contribution, we extend the decidability result to R-automata with Büchi acceptance conditions.

1 Introduction

We consider systems operating on resources which are consumed in small parts and which can be (or have to be) replenished completely at once. To model such systems, we introduce *R-automata* – finite state machines extended by a finite number of unbounded counters corresponding to the resources. The counters can be incremented, reset to zero, or left unchanged along the transitions. When the value of a counter is equal to zero then the stock of this resource is full. Incrementing a counter means using one unit of the resource and resetting a counter means the full replenishment of the stock.

We define the language accepted by an R-automaton relative to a natural number D as the set of words allowing an accepting run of the automaton such that no counter value exceeds D in any state along the run. We study the problem of whether there is a number D such that the corresponding language is universal. This problem corresponds to the fact that with stock size D, the system can exhibit all the behaviors without running out of resources. We show that this problem is decidable in 2-EXPSPACE. As a second technical contribution, we extend the decidability result to R-automata with Büchi acceptance conditions.

* This work has been partially supported by the EU CREDO project.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 67–81, 2008.

To prove decidability of the universality problem, we adopt the technique from [12] and extend it to our setting. We reformulate the problem in the language of finite monoids and solve it using the factorization forest theorem [11]. In [12], this theorem is used for solving the limitedness problem for distance automata. Distance automata are a subclass of R-automata with only one counter which is never reset. In contrast to this model, we handle several counters and resets. This extension cannot be encoded into the distance automata.

The decision algorithm deals with abstractions of collections of runs in order to find and analyze the loops created by these collections. The main step in the correctness proof is to show that each collection of runs along the same word can be split (factorized) into short repeated loops, possibly nested. Having such a factorization, one can analyze all the loops to check that none of the counters is only increased without being reset along them. If none of the counters is increased without being reset then we can bound the counter values by a constant derived from the length of the loops. Since the length of the loops is bounded by a constant derived from the automaton, all words can be accepted by a run with bounded counters. Otherwise, we show that there is a $+$-free regular expression such that for any bound there is a word obtained by pumping this regular expression which does not belong to the language. Therefore, the language cannot be universal for any D.

Related work. The concept of distance automata and the limitedness problem were introduced by Hashiguchi [6]. The limitedness problem is to decide whether there is a natural number D such that all the accepted words can also be accepted with the counter value smaller than D. Different proofs of the decidability of the limitedness problem are reported in [7,10,12]. The last of these results [12] is based on the factorization forest theorem [11,4]. The model of R-automata, which we consider in this paper, extends that of distance automata by introducing resets and by allowing several counters. Furthermore, all the works mentioned above only consider the limitedness problem on finite words, while we here extend the decidability result of the universality problem to the case of infinite words. Distance automata were extended in [8] with additional counters which can be reset following a hierarchical discipline resembling parity acceptance conditions. R-automata relax this discipline and allow the counters to be reset arbitrarily. Universality of a similar type of automata for tree languages is studied in [5]. A model with counters which can be incremented and reset in the same way as in R-automata, called B-automata, is presented in [3]. B-automata accept infinite words such that the counters are bounded along an infinite accepting computation. Decidability of our problems can be obtained using the results from [3]. However, this would require complementation of a B-automaton which results in a non-elementary blowup of the automaton state space.

The fact that R-automata can have several counters which can be reset allows, for instance, to capture the abstractions of the sampled semantics of timed automata [9,1]. A sampled semantics given by a sampling rate $\epsilon = 1/f$ for some positive integer f allows time to pass only in steps equal to multiples of ϵ. The number of different clock valuations within one clock region (a bounded set of

valuations) corresponds to a resource. It is finite for any ϵ while infinite in the standard (dense time) semantics of timed automata. Timed automata can generate runs along which clocks are forced to take different values from the same clock region (an increment of a counter), take exactly the same value (a counter is left unchanged), or forget about the previously taken values (a counter reset).

2 Preliminaries

First, we introduce the model of R-automata and its unparameterized semantics. Then, we introduce the parameterized semantics, the languages accepted by the automaton, and the universality problem.

R-automata. R-automata are finite state machines extended with counters. A transition may increase the value of a counter, leave it unchanged, or reset it back to zero. The automaton on its own does not have the capability of testing the values of the counters. However, the semantics of these automata is parameterized by a natural number D which defines an upper bound on counter values which may appear along the computations of the automaton. Let $\mathbb{N}$ denote the set of non-negative integers.

An *R-automaton* with n counters is a 5-tuple $A = \langle S, \Sigma, \Delta, s_0, F \rangle$ where

- S is a finite set of states,
- Σ is a finite alphabet,
- $\Delta \subseteq S \times \Sigma \times \{0, 1, r\}^n \times S$ is a transition relation,
- $s_0 \in S$ is an initial state, and
- $F \subseteq S$ is a set of final states.

Transitions are labeled (together with a letter) by an effect on the counters. The symbol 0 corresponds to leaving the counter value unchanged, the symbol 1 represents an increment, and the symbol r represents a reset. We use $t, t_1, \ldots$ to denote elements of $\{0, 1, r\}^n$ which we call *effects*. A *path* is a sequences of transitions $(s_1, a_1, t_1, s_2), (s_2, a_2, t_2, s_3), \ldots, (s_m, a_m, t_m, s_{m+1})$, such that $\forall 1 \leq i \leq m.(s_i, a_i, t_i, s_{i+1}) \in \Delta$. An example of an R-automaton is given in Figure 1.

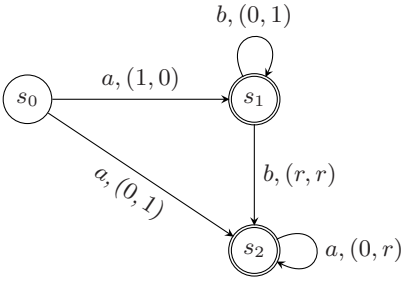


Fig. 1. An R-automaton with two counters

Unparameterized semantics. We define an operation $\oplus$ on the counter values as follows: for any $k \in \mathbb{N}$, $k \oplus 0 = k$, $k \oplus 1 = k+1$, and $k \oplus r = 0$. We extend this operation to n-tuples by applying it componentwise. The operational semantics of an R-automaton $A = \langle S, \Sigma, \Delta, s_0, F \rangle$ is given by a labeled transition system (LTS) $[\![A]\!] = \langle \hat{S}, \Sigma, T, \hat{s_0} \rangle$, where the set of states $\hat{S}$ contains pairs $\langle s, (c_1, \ldots, c_n) \rangle$, $s \in S, c_i \in \mathbb{N}$ for all $1 \leq i \leq n$, with the initial state $\hat{s_0} = \langle s_0, (0, \ldots, 0) \rangle$. The transition relation is defined by $(\langle s, (c_1, \ldots, c_n) \rangle, a, \langle s, (c'_1, \ldots, c'_n) \rangle) \in T$ if and only if $\langle s, a, t, s' \rangle \in \Delta$ and $(c'_1, \ldots, c'_n) = (c_1, \ldots, c_n) \oplus t$. We shall call the states of the LTS *configurations*.

We write $\langle s, (c_1, \ldots, c_n) \rangle \xrightarrow{a} \langle s, (c'_1, \ldots, c'_n) \rangle$ if $(\langle s, (c_1, \ldots, c_n) \rangle, a, \langle s, (c'_1, \ldots, c'_n) \rangle) \in T$. We extend this notation also for words, $\langle s, (c_1, \ldots, c_n) \rangle \xrightarrow{w} \langle s, (c'_1, \ldots, c'_n) \rangle$, where $w \in \Sigma^+$.

Paths in an LTS are called *runs* to distinguish them from paths in the underlying R-automaton. Observe that the LTS contains infinitely many states, but the counter values do not influence the computations, since they are not tested anywhere. In fact, for any R-automaton A, $[\![A]\!]$ is bisimilar to A considered as a finite automaton (without counters and effects). The LTS induced by the R-automaton from Figure 1 is in Figure 2.

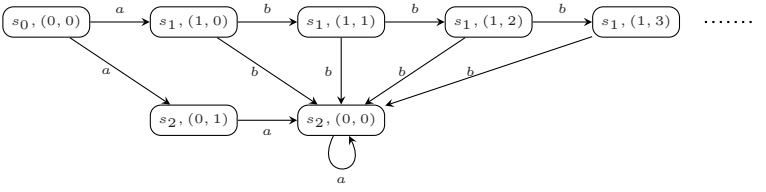


Fig. 2. The unparameterized semantics of the R-automaton in Figure 1

Parameterized Semantics. Now we define the D-semantics of R-automata. We assume that the resources associated to the counters are not infinite and we can use them only for a bounded number of times before they are replenished again. If a machine tries to use a resource which is already completely used up, it is blocked and cannot continue its computation.

For a given $D \in \mathbb{N}$, let $\hat{S}_D$ be the set of configurations restricted to the configurations which do not contain a counter exceeding D, i.e., $\hat{S}_D = \{\langle s, (c_1, \ldots, c_n) \rangle | \langle s, (c_1, \ldots, c_n) \rangle \in \hat{S}$ and $(c_1, \ldots, c_n) \leq (D, \ldots, D)\}$ ($\leq$ is applied componentwise). For an R-automaton A, the *D-semantics* of A, denoted by $[\![A]\!]_D$, is $[\![A]\!]$ restricted to $\hat{S}_D$. We write $\langle s, (c_1, \ldots, c_n) \rangle \xrightarrow{a}_D \langle s, (c'_1, \ldots, c'_n) \rangle$ to denote the transition relation of $[\![A]\!]_D$. We extend this notation for words, $\langle s, (c_1, \ldots, c_n) \rangle \xrightarrow{w}_D \langle s, (c'_1, \ldots, c'_n) \rangle$ where $w \in \Sigma^+$. The 2-semantics of the R-automaton from Figure 1 is in Figure 3.

We abuse the notation to avoid stating the counter values explicitly when it is not necessary. We define the reachability relations $\longrightarrow$ and $\longrightarrow_D$ over pairs of states and words as follows. For $s, s' \in S$ and $w \in \Sigma^+$, $s \xrightarrow{w} s'$ if and only

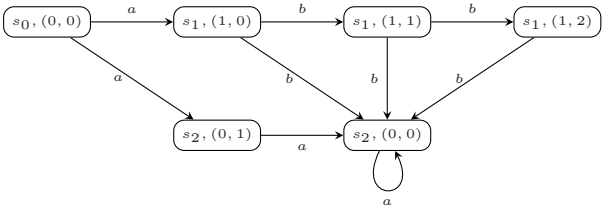


Fig. 3. The 2-semantics of the R-automaton in Figure 1

if there is a path (s, a_1, t_1, s_1), (s_1, a_2, t_2, s_2), $\ldots$, $(s_{|w|-1}, a_{|w|}, t_{|w|}, s')$ such that $w = a_1 \cdot a_2 \cdots a_{|w|}$. For each $D \in \mathbb{N}$, $s \xrightarrow{w}_D s'$ if also for all $1 \leq i \leq |w|$, $t_1 \oplus t_2 \oplus \cdots \oplus t_i \leq (D, \ldots, D)$. It also holds that $s \xrightarrow{w}_D s'$ if and only if there is a run $\langle s, (0, \ldots, 0)\rangle \xrightarrow{w}_D \langle s', (c_1, \ldots, c_n)\rangle$.

Language. The (unparameterized or D-) language of an R-automaton is the set of words which can be read along the runs in the corresponding LTS ending in an accepting state (in a configuration whose first component is an accepting state). The *unparameterized language* accepted by an R-automaton A is $L(A) = \{w | s_0 \xrightarrow{w} s_f, s_f \in F\}$. For a given $D \in \mathbb{N}$, the *D-language* accepted by an R-automaton A is $L_D(A) = \{w | s_0 \xrightarrow{w}_D s_f, s_f \in F\}$. The unparameterized language of the R-automaton from Figure 1 is ab^*a^*. The 2-language of this automaton is $a(\epsilon + b + bb + bbb)a^*$.

Problem Definition. Now we can ask a question about language universality of an R-automaton A parameterized by D, i.e., is there a natural number D such that $L_D(A) = \Sigma^*$. Figure 4 shows an R-automaton A such that $L_2(A) = \Sigma^*$.

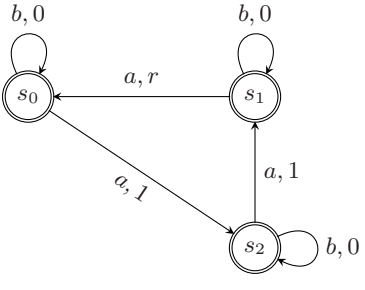


Fig. 4. A 2-universal R-automaton

The language definitions and the universality question can also be formulated for infinite words with Büchi acceptance conditions. The unparameterized ω-language of the automaton from Figure 1 is $ab^\omega + ab^*a^\omega$. The 2-ω-language of this automaton is $a(\epsilon + b + bb + bbb)a^\omega$.

3 Universality

The main result of the paper is the decidability of the universality problem for R-automata formulated in the following theorem.

Theorem 1. *For a given R-automaton A, the question whether there is $D \in \mathbb{N}$ such that $L_D(A) = \Sigma^*$ is decidable in 2-EXPSPACE.*

First, we introduce and also formally define the necessary concepts (patterns, factorization, and reduction) together with an overview of the whole proof. Then we show the construction of the reduced factorization trees and state the correctness of this construction. Finally, we present an algorithm for deciding universality. All proofs can be found in the full version of this paper [2].

3.1 Concepts and Proof Overview

When an R-automaton A is not universal for all $D \in \mathbb{N}$ then there is an infinite set X of words such that for each $D \in \mathbb{N}$ there is $w_D \in X$ and $w_D \notin L_D(A)$. We say then that X is a counterexample. The main step of the proof is to show that there is an X which can be characterized by a $+$-free regular expression. In fact, we show that X also satisfies a number of additional properties which enable us to decide for every such a $+$-free regular expression, whether it corresponds to a counterexample or not. Another step of the proof is to show that we need to check only finitely many such $+$-free regular expressions in order to decide whether there is a counterexample at all.

Patterns. The standard procedure for checking universality in the case of finite automata is subset construction. Whenever there are non-deterministic transitions $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$ then we build a "summary" transition $\{s\} \xrightarrow{a} \{s_1, s_2\}$. This summary transition says that from the set of states $\{s\}$ we get to the set of states $\{s_1, s_2\}$ after reading the letter a. In the case of R-automata, subset construction is in general not guaranteed to terminate since the values of the counters might grow unboundedly. To deal with this problem, we exploit the fact that the values of the counters do not influence the computations of the automaton. Therefore, we perform an abstraction which hides the actual values of the counters and considers only the effects along the transitions instead. The abstraction leads to a more complicated variant of summary transitions namely so called *patterns*.

We define a commutative, associative, and idempotent operation $\circ$ on the set $\{0, 1, r\}$: $0 \circ 0 = 0$, $0 \circ 1 = 1$, $0 \circ r = r$, $1 \circ 1 = 1$, $1 \circ r = r$, and $r \circ r = r$. In fact, if we define an order $0 < 1 < r$ then $\circ$ is the operation of taking the maximum. We extend this operation to effects, i.e., n-tuples, by applying it componentwise (this preserves all the properties of $\circ$). An effect obtained by adding several other effects through the application of the operator $\circ$ summarizes the manner in which the counters are changed. More precisely, it describes whether a counter is reset or whether it is increased but not reset or whether it is only left untouched.

A pattern $\sigma : (S \times S) \longrightarrow 2^{\{0,1,r\}^n}$ is a function from pairs of automaton states to sets of effects. Let us denote patterns by $\sigma, \sigma_1, \sigma', \ldots$. As an example, consider a pattern σ involving states s and s' and two counters. Let $\sigma(s,s) = \{(0,0),(1,1)\}$, $\sigma(s',s') = \{(1,1),(1,0)\}$, $\sigma(s,s') = \{(1,1)\}$ and $\sigma(s',s) = \{(1,1)\}$. This pattern is depicted in Figure 5a.

Clearly, for a given R-automaton there are only finitely many patterns; let us denote this finite set of all patterns by $\mathbb{P}$. We define an operation $\bullet$ on $\mathbb{P}$ as follows. Let $(\sigma_1 \bullet \sigma_2)(s,s') = \{t \mid \exists s'', t_1, t_2.\ t_1 \in \sigma_1(s,s''), t_2 \in \sigma_2(s'',s'), t = t_1 \circ t_2\}$. Note, that $\bullet$ is associative and it has a unit σ_e, where $\sigma_e(s,s') = \{(0,\ldots,0)\}$ if $s = s'$ and $\sigma_e(s,s') = \emptyset$ otherwise. Therefore, $(\mathbb{P}, \bullet)$ is a finite monoid.

For each word we obtain a pattern by running the R-automaton along this word. Formally, let $\mathsf{Run} : \Sigma^+ \longrightarrow \mathbb{P}$ be a homomorphism defined by $\mathsf{Run}(a) = \sigma$, where $t \in \sigma(s,s')$ if and only if $(s,a,t,s') \in \Delta$.

Loops. In the case of finite automata, a set of states L and a word w constitute a loop in the subset construction if $L \xrightarrow{w} L$, i.e., starting from L and reading w, we end up in L again. The intuition behind the concept of a loop is that several iterations of the loop have the same effect as a single iteration. In our abstraction using patterns, loops are words w such that w yields the same pattern as $w^2, w^3, \ldots$. We can skip the starting set of states, because the function Run starts implicitly from the whole set of states S (if there are no runs between some states then the corresponding set of effects is empty). More precisely, a word w is a loop if $\mathsf{Run}(w)$ is an idempotent element of the pattern monoid. Two loops are identical if they produce the same pattern. Observe that the pattern in Figure 5a is idempotent.

Factorization. We show that each word can be split into *short identical loops* repeated many times. The loops can possibly be nested, so that this split (*factorization*) defines a factorization tree. The idea is that since we have such a factorization for each word, it is sufficient to analyze only the (short) loops and either find a run with bounded maximal value of the counters or use the loop structure to construct a counterexample regular expression.

On a higher level we can see a factorization of words as a function which for every word $w = a_1 a_2 \cdots a_l$ returns its factorization tree, i.e., a finite tree with branching degree at least 2 (except for the leaves) and with nodes labeled by subwords of w such that the labeling function satisfies the following conditions:

- if a node labeled by v has children labeled by $w_1, w_2, \ldots, w_m$ then $v = w_1 \cdot w_2 \cdots w_m$,
- if $m \geq 3$ then $\sigma = \mathsf{Run}(v) = \mathsf{Run}(w_i)$ for all $1 \leq i \leq m$ and σ is idempotent,
- the leaves are labeled by $a_1, a_2, \ldots, a_l$ from left to right.

An example of such a tree is in Figure 5b. It follows from the factorization forest theorem [11,4] that there is such a (total) function which returns trees whose height is bounded by $3 \cdot |\mathbb{P}|$ where $|\mathbb{P}|$ is the size of the monoid.

We define the length of a loop as the length of the word (or a pattern sequence) provided that only the two longest iterations of the nested loops are counted.

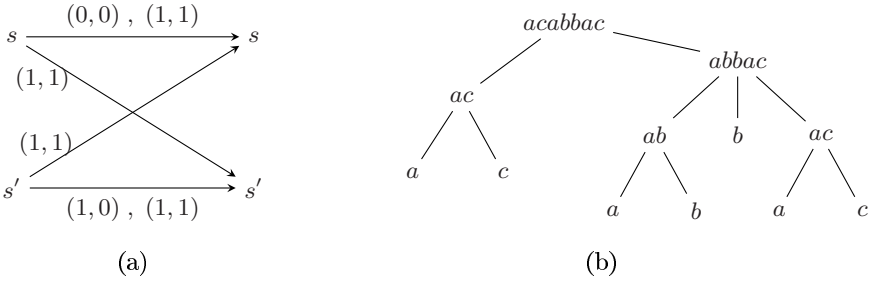


Fig. 5. A pattern involving two states and two counters (a) and a factorization tree (b). $\mathsf{Run}(abbac) = \mathsf{Run}(ab) = \mathsf{Run}(b) = \mathsf{Run}(ac)$ and it is idempotent.

This concept is defined formally in Subsection 3.3. We say that the loops are short if there is a bound given by the automaton so that the length of all the loops is shorter than this bound. A consequence of the factorization forest theorem is that there is a factorization such that all loops are short.

Reduction. We have defined the loops so that the iterations of a loop have the same effect as the loop itself. Therefore, it is enough to analyze a single iteration to tell how the computations look when the loop is iterated an arbitrary number of times. By a *part* in an idempotent pattern σ, we mean an element (an effect) in the set $\sigma(s, s')$ for some states s and s'. We will distinguish between two types of parts, namely *bad* and *good* parts. A bad part corresponds only to runs along which the increase of some counter is at least as big as the number of the iterations of the loop. A part is *good* if there is a run with this effect along which the increase is bounded by the maximal increase induced by two iterations of the loop. Formally, we define a function reduce which for each pattern returns a pattern containing all good parts of the original pattern, but no bad part. Then we illustrate it on a number of examples.

For a pattern σ, $\mathsf{core}(\sigma)$ is defined as follows:

$$\mathsf{core}(\sigma)(s, s') = \begin{cases} \sigma(s, s') \cap \{0, r\}^n & \text{if } s = s' \\ \emptyset & \text{otherwise} \end{cases}$$

Let $\mathsf{reduce}(\sigma) = \sigma \bullet \mathsf{core}(\sigma) \bullet \sigma$.

For an automaton with one state s, one counter, and a loop w with pattern σ, if $\sigma(s, s) = \{(1)\}$ then the whole pattern is bad, i.e., $\mathsf{reduce}(\sigma)(s, s) = \emptyset$. Notice that any run over w^k increases the counter by k. On the other hand, if $\sigma(s, s) = \{(0)\}$ or $\sigma(s, s) = \{(r)\}$ then the whole pattern is good, i.e., $\mathsf{reduce}(\sigma) = \sigma$.

With more complicated patterns we need a more careful analysis. Let us consider a loop w with pattern σ where $\sigma(s, s) = \{(0)\}$, $\sigma(s', s') = \{(1)\}$, $\sigma(s, s') = \{(1)\}$, and $\sigma(s', s) = \{(1)\}$. We will motivate why the part $(1) \in \sigma(s', s')$ is good. For any k, we can take the run over w^k which starts from s', moves to s after the first iteration, stays in s for $k - 2$ iterations, and finally moves back to s' after the k^{th} iteration. Then, the effect of the run is (1). Furthermore, the counter

increase along the run is bounded by twice the maximal counter increase while reading w. In fact, using a similar reasoning, we can show that all parts of σ are good (which is consistent with the fact that $\mathsf{reduce}(\sigma) = \sigma$).

As the last example, let us consider the pattern from Figure 5a. First, we show that the part $(1, 0) \in \sigma(s', s')$ is bad. The only run over w^k with effect $(1, 0)$ is the one which comes back to s' after each iteration. However, this run increases the first counter by k. On the other hand, the part $(1, 1) \in \sigma(s', s')$ is good by a similar reasoning to the previous example. In fact, we can show that all other parts of the pattern are good (which is consistent with the value of $\mathsf{reduce}(\sigma)$ in Figure 6).

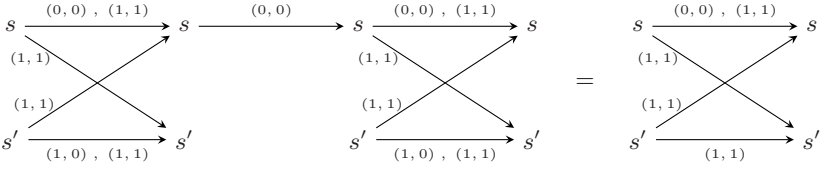


Fig. 6. $\sigma \bullet \mathsf{core}(\sigma) \bullet \sigma = \mathsf{reduce}(\sigma)$ where σ is the pattern from Figure 5a

Reduced Factorization Trees. For a factorization of a word w, we need to check whether there is a run which goes through a good part in every loop. In order to do that, we enrich the tree structure, so that each node will now be labeled, in addition to a word, also by a pattern. The patterns are added by the following function: given an input sequence of patterns, the leaves are labeled by the elements of the sequence, nodes with branching degree 2 are labeled by the composition of the children labels, and we label each node with branching degree at least 3 by σ, where σ is the idempotent label of all its children. Now, based on this labeling, we build a *reduced factorization tree* for w in several steps (formally described in Subsection 3.2).

We start with the sequence of patterns obtained by Run from the letters of the word. In each step, we take the resulting sequence from the previous step, build a factorization tree from it, and label it by patterns as described above. Then we take the lowest nodes such that they have at least 3 children and they are labeled by a pattern σ such that $\mathsf{reduce}(\sigma) \neq \sigma$. We change the labels of these nodes to $\mathsf{reduce}(\sigma)$. We pack the subtrees of these nodes into elements of the new sequence and we leave other elements of the sequence unmodified. This procedure eventually terminates and returns one tree with the following properties (the important invariant is shown in Lemma 1):

- if a node labeled by σ has two children labeled by σ_1, σ_2 then $\sigma = \sigma_1 \bullet \sigma_2$,
- if a node labeled by σ has m children labeled by $\sigma_1, \ldots, \sigma_m$, $m \geq 3$, then $\sigma_i = \sigma_j$ for all $1 \leq i, j \leq m$, σ_1 is idempotent, and $\sigma = \mathsf{reduce}(\sigma_1)$.

An example of a reduced factorization tree is in Figure 7. We show that there is a factorization function such that the height of all reduced factorization trees produced by it is bounded by $3 \cdot |\mathbb{P}|^2$ (Lemma 2) using the factorization forest

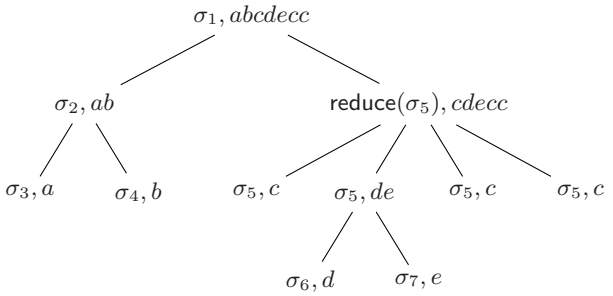


Fig. 7. An example reduced factorization tree. $\sigma_1 = \sigma_2 \bullet \mathsf{reduce}(\sigma_5), \sigma_2 = \sigma_3 \bullet \sigma_4$, and $\sigma_5 = \sigma_6 \bullet \sigma_7$. For all leaves labeled by $\hat{\sigma}, \hat{a}$, $\hat{\sigma} = \mathsf{Run}(\hat{a})$.

theorem and a property of the reduction function that if $\mathsf{reduce}(\sigma) \neq \sigma$ then $\mathsf{reduce}(\sigma) <_{\mathcal{J}} \sigma$, where $<_{\mathcal{J}}$ is the usual ordering of the $\mathcal{J}$-classes on $\mathbb{P}$, $\mathcal{J}$ is a standard Green's relation; $\sigma \leq_{\mathcal{J}} \sigma'$ if and only if there are σ_1, σ_2 such that $\sigma = \sigma_1 \bullet \sigma' \bullet \sigma_2$; $\sigma <_{\mathcal{J}} \sigma'$ if and only if $\sigma \leq_{\mathcal{J}} \sigma'$ and $\sigma' \not\leq_{\mathcal{J}} \sigma$ (Lemma 2 in [2]).

Correctness. Let σ be the label of the root of a reduced factorization tree for a word w and let $\mathsf{pump}(r, k)$ for a $+$-free regular expression r and for a $k \in \mathbb{N}$ be the word obtained by repeating each r_1, where r_1^* is a subexpression of r, k-times. Then

- if $\sigma(s_0, s_f) \neq \emptyset$ for some $s_f \in F$ then there is a run from s_0 to s over w in $8^{|\mathbb{P}|^2}$-semantics,
- otherwise, there is a $+$-free regular expression r such that for all D there is a k such that there is a counter which exceeds D along all runs from s_0 to s_f, $s_f \in F$, over $\mathsf{pump}(r, k)$.

The previous items are formulated in Subsection 3.3, Lemma 4 and Lemma 5.

Relation to Simon's Approach. There are several important differences between the method presented in this paper and that of Simon [12]. Our notion of pattern is a function to a set of effects, while in Simon's case it is a function to the set $\{0, 1, \omega\}$. Because of the resets and the fact that there are several counters, it is not possible to linearly order the effects. Thus, a collection of automaton runs can be abstracted into several incomparable effects. The sets are necessary in order to remember all of them. Furthermore, the different notion of pattern requires a new notion of reduction which does not remove loops labeled also by resets. We need to show then that application of this notion of reduction during the construction of the reduced factorization trees preserves the correctness.

3.2 Construction of the Reduced Factorization Tree

We define labeled finite trees to capture the looping structure of pattern sequences. Let Γ be a set of finite trees with two labeling functions Pat and Word, which for each node return a pattern and a word, respectively. We will abuse

the notation and, for a tree T, we use $\mathsf{Pat}(T)$ or $\mathsf{Word}(T)$ to denote $\mathsf{Pat}(N)$ or $\mathsf{Word}(N)$, respectively, where N is the root of T. We also identify nodes with the subtrees in which they are roots. We can then say that a node T has children $T_1, \ldots, T_m$ and then use T_i's as trees. For a tree T, we define its height $h(T)$ as $h(T) = 1$ if T is a leaf, $h(T) = 1 + \max\{h(T_1), \ldots, h(T_m)\}$ if $T_1, \ldots, T_m$ are children of the root of T.

By Γ^+ we mean the set of nonempty sequences of elements of Γ. By $(\Gamma^+)^+$ we mean the set of nonempty sequences of elements of Γ^+. Let us denote elements of Γ^+ by $\gamma, \gamma_1, \gamma', \ldots$. For $\gamma \in \Gamma^+$, let $|\gamma|$ denote the length of γ.

Let $f : \Gamma^+ \to \mathbb{P}$ be a homomorphism with respect to $\bullet$ defined by $f(T) = \mathsf{Pat}(T)$. We call a function $d : \Gamma^+ \to (\Gamma^+)^+$ a *factorization function* if it satisfies the following conditions. If $d(\gamma) = (\gamma_1, \gamma_2, \ldots, \gamma_m)$ then $\gamma = \gamma_1 \cdot \gamma_2 \cdots \gamma_m$, if $m = 1$ then $|\gamma| = 1$, and if $m \geq 3$ then $f(\gamma) = f(\gamma_i)$ for all $1 \leq i \leq m$ and $f(\gamma)$ is an idempotent element.

For a factorization function d we define two functions $\mathsf{tree} : \Gamma^+ \to \Gamma$ and $\mathsf{cons} : \Gamma^+ \to \Gamma^+$ inductively as follows. Let $\langle \sigma, w \rangle$ denote a tree which consists of only the root labeled by σ and w.

$$\mathsf{tree}(\gamma) = \begin{cases} \gamma & \text{if } |\gamma| = 1, \\ \langle \sigma_1 \bullet \sigma_2, w_1 \cdot w_2 \rangle \text{ with children } \mathsf{tree}(\gamma_1), \mathsf{tree}(\gamma_2), & \text{if } d(\gamma) = (\gamma_1, \gamma_2), \\ \qquad \sigma_i = \mathsf{Pat}(\mathsf{tree}(\gamma_i)), w_i = \mathsf{Word}(\mathsf{tree}(\gamma_i)) \text{ for } i \in \{1, 2\}, & \\ \langle \mathsf{reduce}(\sigma), w_1 \cdot w_2 \cdots w_m \rangle \text{ with children } \mathsf{tree}(\gamma_1), \ldots, \mathsf{tree}(\gamma_m), & \text{if} \\ \qquad m \geq 3, d(\gamma) = (\gamma_1, \gamma_2, \ldots, \gamma_m), \sigma = \mathsf{Pat}(\mathsf{tree}(\gamma_1)), & \\ \qquad \text{and } w_i = \mathsf{Word}(\mathsf{tree}(\gamma_i)) \text{ for all } 1 \leq i \leq m. & \end{cases}$$

The function tree builds a tree (resembling a factorization tree) from the sequence of trees according to the function d. The only difference from straightforwardly following the function d is that the labeling function Pat might be changed by the function reduce. Let us color the trees in the function cons either green or red during the inductive construction of a new sequence.

$$\mathsf{cons}(\gamma) = \begin{cases} \gamma & \text{if } |\gamma| = 1. \text{ Mark } \gamma \text{ green.} \\ \mathsf{cons}(\gamma_1) \cdot \mathsf{cons}(\gamma_2) \cdots \mathsf{cons}(\gamma_m) & \\ & \text{if } d(\gamma) = (\gamma_1, \gamma_2, \ldots, \gamma_m) \text{ and either } m = 2 \text{ or} \\ & \text{there is } 1 \leq i \leq m \text{ such that } \mathsf{cons}(\gamma_i) \text{ contains} \\ & \text{a red tree or } \mathsf{reduce}(f(\gamma_1)) = f(\gamma_1). \\ \mathsf{tree}(\gamma) & \text{if } d(\gamma) = (\gamma_1, \gamma_2, \ldots, \gamma_m), m \geq 3, \text{ no } \mathsf{cons}(\gamma_i) \\ & \text{contains a red tree and } \mathsf{reduce}(f(\gamma_1)) \neq f(\gamma_1). \\ & \text{Mark the tree red.} \end{cases}$$

The function cons updates the sequence of trees trying to leave as much as possible untouched, but whenever Pat would be changed by the reduce function for the first time (on the lowest level), it packs the whole sequence into a single tree with changed Pat label of the root using the function tree.

The important property of the construction is that for each tree in the new sequence it holds that whenever a node has more than two children, they are all labeled by identical idempotent patterns. Let us call a tree *balanced* if whenever

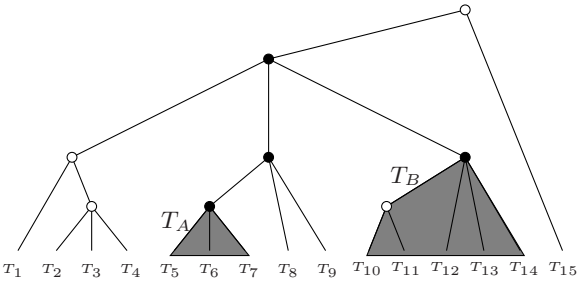


Fig. 8. Application of cons to $T_1 \cdots T_{15}$. The black nodes represent the nodes for which $\mathsf{reduce}(\sigma) \neq \sigma$. The resulting sequence is $T_1T_2T_3T_4T_AT_8T_9T_BT_{15}$.

a node T has children $T_1, T_2, \ldots, T_m$, where $m \geq 3$, then $\mathsf{Pat}(T_1) = \mathsf{Pat}(T_2) = \cdots = \mathsf{Pat}(T_m)$, it is an idempotent element in $\mathbb{P}$, and $\mathsf{Pat}(T) = \mathsf{reduce}(\mathsf{Pat}(T_1))$.

Lemma 1. *For a $\gamma \in \Gamma^+$, if all trees in γ are balanced then all trees in $\mathsf{cons}(\gamma)$ are balanced.*

Now we show how to get a sequence of trees from runs of the automaton. Let $\mathsf{treeRun} : \Sigma^+ \longrightarrow \Gamma^+$ be a homomorphism with respect to the word composition defined by $\mathsf{treeRun}(a) = \langle \mathsf{Run}(a), a \rangle$.

Assume that there is a factorization function d fixed. Let for a word $w \in \Sigma^+$, γ_w be defined as $\mathsf{cons}^n(\mathsf{treeRun}(w))$, where $n \in \mathbb{N}$ is the least such that $\mathsf{cons}^n(\mathsf{treeRun}(w)) = \mathsf{cons}^{n+1}(\mathsf{treeRun}(w))$. Note that γ_w is always defined, because for all $\gamma \in \Gamma^+$, $|\mathsf{cons}(\gamma)| \leq |\gamma|$ and if $|\mathsf{cons}(\gamma)| = |\gamma|$ then $\mathsf{cons}(\gamma) = \gamma$. Let $T_w = \mathsf{tree}(\gamma_w)$. We call T_w the *reduced factorization tree* of w constructed by d. From Lemma 1 it follows that T_w is balanced (note that if $\mathsf{cons}^n(\gamma) = \mathsf{cons}^{n+1}(\gamma)$ then $\mathsf{cons}^n(\gamma)$ contains only green trees).

Remark. Notice that we do not explicitly mention the factorization function d in the definition of a reduced factorization tree T_w constructed by d from a word w. It is always clear from the context which factorization function we mean.

We show that for each R-automaton there is a factorization function such that for any w the height of the tree T_w is bounded by a constant computed from the parameters of the automaton.

Lemma 2. *Given an R-automaton A, there is a factorization function d such that for all words $w \in \Sigma^+$, $h(T_w) \leq 3 \cdot |\mathbb{P}|^2$.*

3.3 Correctness

To formulate the first correctness lemma, we define the following concept of a length function $l : \Gamma \to \mathbb{N}$ inductively by

$$l(T) = \begin{cases} 1 & \text{if } T \text{ is a leaf} \\ l(T_1) + l(T_2) & \text{if } T \text{ has two children } T_1, T_2 \\ 2 \cdot \max\{l(T_1), \ldots, l(T_m)\} & \text{if } T \text{ has children } T_1, \ldots, T_m, m \geq 3 \end{cases}$$

By induction on $h(T_w)$ and using the bound derived in Lemma 2, one can show the following claim.

Lemma 3. *Given an R-automaton A, there is a factorization function d such that for all words $w \in \Sigma^+$, $l(T_w) \leq 8^{|\mathbb{P}|^2}$.*

We say that $s \xrightarrow{w} s'$ or $s \xrightarrow{w}_D s'$ *realizes* t if there is a witnessing path $(s, a_1, t_1, s_1), (s_1, a_2, t_2, s_2), \ldots, (s_{|w|-1}, a_{|w|}, t_{|w|}, s')$ such that $t = t_1 \circ t_2 \circ \cdots \circ t_{|w|}$.

Let us define $\mathsf{Run}_D(w)$ to be the pattern obtained by running the automaton over w in the D-semantics. Formally, $\mathsf{Run}_D(w)(s, s')$ contains t if and only if $s \xrightarrow{w}_D s'$ realizes t. Note that the function Run_D is not a homomorphism with respect to the word composition. We also define a relation $\sqsubseteq$ on patterns by $\sigma \sqsubseteq \sigma'$ if and only if for all s, s', $\sigma(s, s') \subseteq \sigma'(s, s')$.

From Lemma 3 we show that there is a factorization function such that for every w, $\mathsf{Pat}(T_w)$ corresponds to the runs of the R-automaton which can be performed in the D-semantics for any big enough D. This is formulated in the following lemma.

Lemma 4. *Given an R-automaton, there is a factorization function such that for all $w \in \Sigma^+$ and for all $D \in \mathbb{N}, D \geq 8^{|\mathbb{P}|^2}$, $\mathsf{Pat}(T_w) \sqsubseteq \mathsf{Run}_D(w)$.*

Of particular interest are runs starting in the initial state.

Corollary 1. *Given an R-automaton A, there is a factorization function such that for all words w, if $\mathsf{Pat}(T_w)(s_0, s) \neq \emptyset$ then there is a run $\langle s_0, (0, \ldots, 0)\rangle \xrightarrow{w}_D \langle s, (c_1, \ldots, c_n)\rangle$ where $D = l(T_w)$.*

It remains to show that if the relation between the patterns in the previous lemma is strict then there is a word for each D which is a witness for the strictness, i.e., the runs over this word in the D-semantics generate a smaller pattern than over the original word. These witness words are generated from a $+$-free regular expression r by pumping r_1 for all subexpressions r_1^* of r. Let us define a function re which for a reduced factorization tree returns a $+$-free regular expression inductively by

$$\mathsf{re}(T) = \begin{cases} \mathsf{Word}(T) & \text{if } T \text{ is a leaf} \\ \mathsf{re}(T_1) \cdot \mathsf{re}(T_2) & \text{if } T \text{ has two children } T_1, T_2 \\ (\mathsf{re}(T_1))^* & \text{if } T \text{ has children } T_1, T_2, \ldots, T_m, m \geq 3 \end{cases}$$

For a $+$-free regular expression r and a natural number $k > 0$, let the function $\mathsf{pump}(r, k)$ be defined inductively as follows: $\mathsf{pump}(a, k) = a$, $\mathsf{pump}(r_1 \cdot r_2, k) = \mathsf{pump}(r_1, k) \cdot \mathsf{pump}(r_2, k)$, and $\mathsf{pump}(r^*, k) = \mathsf{pump}(r, k)^k$.

For example, $\mathsf{pump}(a(bc^*d)^*aa^*, 2) = abccdbccdaaa$.

Lemma 5. *Given an R-automaton and a factorization function, for all $w \in \Sigma^+$ and all $D \in \mathbb{N}$ there is a $k \in \mathbb{N}$ such that $\mathsf{Run}_D(\mathsf{pump}(\mathsf{re}(T_w), k)) \sqsubseteq \mathsf{Pat}(T_w)$.*

A special case are runs starting from the initial state.

Corollary 2. *Given an R-automaton, for any $w \in \Sigma^+$, if $\mathsf{Pat}(T_w)(s_0, s) = \emptyset$ then $\forall D \exists k$ such that there is no run $\langle s_0, (0, \ldots, 0)\rangle \xrightarrow{v}_D \langle s, (c_1, \ldots, c_n)\rangle$ where $v = \mathsf{pump}(\mathsf{re}(T_w), k)$.*

3.4 Algorithm

To check the universality of an R-automaton A, we have to check all patterns σ such that $\sigma = \mathsf{Pat}(T_w)$ for some $w \in \Sigma^+$ and some factorization function. If there is a σ such that for all $s_f \in F$, $\sigma(s_0, s_f) = \emptyset$ then for all $D \in \mathbb{N}$, $L_D(A) \neq \Sigma^*$. This gives us the following algorithm. Recall that σ_e denotes the unit of $(\mathbb{P}, \bullet)$.

The algorithm uses a set of patterns P as the data structure. Given an R-automaton $A = \langle S, \Sigma, \Delta, s_0, F \rangle$ on the input, it answers 'YES' or 'NO'. The set P is initialized by $P = \{\sigma | \sigma = \mathsf{Run}(a), a \in \Sigma\} \cup \{\sigma_e\}$.

While $|P|$ increases the algorithm performs the following operations:

- pick $\sigma_1, \sigma_2 \in P$ and add $\sigma_1 \bullet \sigma_2$ back to P.
- pick a $\sigma \in P$ such that σ is idempotent and add $\mathsf{reduce}(\sigma)$ back to P.

If there is $\sigma \in P$ such that for all $s_f \in F$, $\sigma(s_0, s_f) = \emptyset$, answer 'NO', otherwise, answer 'YES'.

The correctness is stated in the following theorem. See [2] for the full proof.

Theorem 2. *The algorithm is correct and runs in 2-EXPSPACE.*

Proof. The algorithm terminates because $\mathbb{P}$ is finite. Its correctness follows from the previous two corollaries. The algorithm needs space $|\mathbb{P}|$ (the number of different patterns). The size of $\mathbb{P}$ is $2^{(3^n)\cdot|S|^2}$ ($|S|^2$ different pairs of states, $2^{(3^n)}$ different sets of effects). Therefore, the algorithm needs double exponential space.

4 Büchi Universality

The universality problem is also decidable for R-automata with Büchi acceptance conditions.

Theorem 3. *For a given R-automaton A, the question whether there is $D \in \mathbb{N}$ such that $L^\omega_D(A) = \Sigma^\omega$ is decidable in 2-EXPSPACE.*

To show this result, we need to extend patterns by accepting state information. A pattern is now a function $\sigma : S \times S \longrightarrow 2^{\{0,1\}\times\{0,1,r\}^n}$, where for s, s' and $\langle a, t \rangle \in \sigma(s, s')$, the value of a encodes whether there is a path from s to s' realizing t which meets an accepting state. For instance, $\sigma(s, s') = \{\langle 0, (0, r)\rangle, \langle 1, (1, 1)\rangle\}$ means that there are two different types of paths between s and s': they either realize $(0, r)$ but do not visit an accepting state, or realize $(1, 1)$ and visit an accepting state. We define the composition $\bullet$ by defining the composition on the accepting state: $0 \circ 0 = 0, 0 \circ 1 = 1 \circ 0 = 1 \circ 1 = 1$. The set of patterns (denote again $\mathbb{P}$) with $\bullet$ is a finite monoid. We define the function reduce in the same way as before, i.e., the accepting state information does not play any role there. Clearly, either $\mathsf{reduce}(\sigma) = \sigma$ or $\mathsf{reduce}(\sigma) <_{\mathcal{J}} \sigma$, so the reduced factorization trees have bounded height. Lemma 4 and Lemma 5 also hold, because (non)visiting an accepting state does not influence the runs in the D-semantics.

This allows us to use the same algorithm as for the finite word universality problem, except for the condition for concluding non-universality. The condition

is whether there are $\sigma_1, \sigma_2 \in P$ such that σ_2 is idempotent and for all s such that $\sigma_1(s_0, s) \neq \emptyset$ the following holds. If $\langle a, t\rangle \in \sigma_2(s, s)$ then either $a = 0$ or $t \notin \{0, r\}^n$. For a full proof of Theorem 3 see [2].

5 Conclusions

We have defined R-automata – finite automata extended with unbounded counters which can be left unchanged, incremented, or reset along the transitions. As the main result, we have shown that the following problem is decidable in 2-EXPSPACE. Given an R-automaton, is there a bound such that all words are accepted by runs along which the counters do not exceed this bound? We have also extended this result to R-automata with Büchi acceptance conditions.

As a future work, one can consider the (bounded) universality or limitedness question to vector addition systems (VASS) or reset vector addition systems (R-VASS), where the latter form a superclass of R-automata. The limitedness problem can be shown undecidable for R-VASS for both finite word and ω-word case, while it is an open question for VASS. The universality problem can be shown to be undecidable for R-VASS for ω-word case, in other cases it is open.

References

1. Abdulla, P.A., Krcal, P., Yi, W.: Sampled universality of timed automata. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 2–16. Springer, Heidelberg (2007)
2. Abdulla, P.A., Krcal, P., Yi, W.: R-automata. Technical Report 2008-015, IT Department, Uppsala University (June 2008)
3. Bojańczyk, M., Colcombet, T.: Bounds in omega-regularity. In: Proc. of LICS 2006, pp. 285–296. IEEE Computer Society Press, Los Alamitos (2006)
4. Colcombet, T.: Factorisation forests for infinite words. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 226–237. Springer, Heidelberg (2007)
5. Colcombet, T., Löding, C.: The non-deterministic Mostowski hierarchy and distance-parity automata. In: Proc. of ICALP 2008. LNCS, vol. 5126, pp. 398–409. Springer, Heidelberg (2008)
6. Hashiguchi, K.: Limitedness theorem on finite automata with distance functions. Journal of Computer and System Sciences 24(2), 233–244 (1982)
7. Hashiguchi, K.: Improved limitedness theorems on finite automata with distance functions. Theoretical Computer Science 72(1), 27–38 (1990)
8. Kirsten, D.: Distance desert automata and the star height problem. Informatique Theorique et Applications 39(3), 455–509 (2005)
9. Krcal, P., Pelanek, R.: On sampled semantics of timed systems. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 310–321. Springer, Heidelberg (2005)
10. Leung, H.: Limitedness theorem on finite automata with distance functions: an algebraic proof. Theoretical Computer Science 81(1), 137–145 (1991)
11. Simon, I.: Factorization forests of finite height. Theoretical Computer Science 72(1), 65–94 (1990)
12. Simon, I.: On semigroups of matrices over the tropical semiring. Informatique Theorique et Applications 28(3-4), 277–294 (1994)

Distributed Timed Automata with Independently Evolving Clocks*

S. Akshay[1,3], Benedikt Bollig[1], Paul Gastin[1],
Madhavan Mukund[2], and K. Narayan Kumar[2]

[1] LSV, ENS Cachan, CNRS, France
{akshay,bollig,gastin}@lsv.ens-cachan.fr
[2] Chennai Mathematical Institute, Chennai, India
{madhavan,kumar}@cmi.ac.in
[3] The Institute of Mathematical Sciences, Chennai, India

Abstract. We propose a model of distributed timed systems where each component is a timed automaton with a set of local clocks that evolve at a rate independent of the clocks of the other components. A clock can be read by any component in the system, but it can only be reset by the automaton it belongs to.

There are two natural semantics for such systems. The *universal* semantics captures behaviors that hold under any choice of clock rates for the individual components. This is a natural choice when checking that a system always satisfies a positive specification. However, to check if a system avoids a negative specification, it is better to use the *existential* semantics—the set of behaviors that the system can possibly exhibit under some choice of clock rates.

We show that the existential semantics always describes a regular set of behaviors. However, in the case of universal semantics, checking emptiness turns out to be undecidable. As an alternative to the universal semantics, we propose a *reactive* semantics that allows us to check positive specifications and yet describes a regular set of behaviors.

1 Introduction

In today's world, it is becoming increasingly important to look at networks of timed systems, which allow real-time systems to operate in a distributed manner. Many real-life systems, such as mobile phones, computer servers, and railway crossings, depend crucially on timing while usually consisting of many interacting systems. In general, there is no reason to assume that different timed systems in the networks refer to the same time or evolve at the same rate.

Timed automata [2] are a well-studied formalism to describe systems that require timing. However, networks of timed automata, under the assumption of knowledge of a global time, as done in [5, 10], do not really reflect the distributed model. In this paper, we provide a framework to look at distributed systems with independently evolving local clocks. Each constituent system is modeled by a timed automaton. All clocks belonging to this timed automaton evolve at the same rate. However clocks belonging

* Partially supported by ARCUS, DOTS (ANR-06-SETIN-003), and P2R MODISTE-COVER/RNP Timed-DISCOVERI.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 82–97, 2008.

to different processes are allowed to evolve at rates that are independent of each other. We allow clocks belonging to one process to be read/checked by another but we require that a clock can only be reset by the automaton it belongs to. In addition, since we have differing time values on different processes, we are interested in the underlying untimed behaviors of these distributed timed automata rather than their timed behaviors. Thus, the clocks (and time itself) are implementation or synchronization tools rather than being a part of the observation. To ensure that we focus on this problem of varying local time rates, we move to a more general setting with shared memory, which allows us to describe more general systems such as networks of timed asynchronous automata.

It is now natural to look at different semantics depending on the specifications that we want our system to satisfy. When we want to guarantee that our system exhibits a positive specification, we look at the *universal* semantics. This semantics describes the behaviors exhibited by the system no matter how time evolves in the constituent processes. However, if we want to check that our system avoids a negative specification, then we prefer to look at the *existential* semantics. This is the set of behaviors that the system might exhibit under some (bad) choice of local time rates in the constituent processes. We perform a region construction on our distributed timed automata to show that the existential semantics always gives a regular set of untimed behaviors. Thus the model checking problem of distributed timed automata against regular negative specifications is decidable as well. On the other hand, we show that checking emptiness for the universal semantics is undecidable. This is done by a reduction from Post's correspondence problem (PCP) by encoding a PCP instance in terms of the local time rates and ensuring that there is a solution to the PCP instance if and only if there is a valid behavior under all local time rates. This result is further strengthened to a bounded case, where we have restrictions on the relative time rates. Finally, to be able to synthesize and check for positive specifications, we introduce a more intuitive *reactive* semantics, which has the additional advantage of ensuring decidability. This model corresponds to being able to make sure, step by step, that a positive specification is exhibited by our system. This is formally done by defining an equivalent alternating automaton, generating a regular behavior.

Related work. In [6, 12], classical timed automata are equipped with an additional parameter Δ, which allows a clock to diverge over a period t from its actual value by Δt. This model conforms, in a sense, to our existential semantics, where we restrict the set of clock rates to those corresponding to Δ (see Section 5). Syntactically, our model coincides with that from [7]: A clock can only be reset by the owner process, whereas it can be read by any process. However, the above works differ from ours since they consider timed words rather than untimed languages. This also explains why our automata differ from hybrid automata [9]. In the model of [3], clocks are not shared and clocks on different processes drift only as long as the processes do not communicate. These assumptions make partial-order–reduction techniques applicable. A fundamental difference between all these approaches and our work is that we do not restrict to system configurations that can be reached under *some* local-time behavior. We also tackle the problem of checking positive specifications by providing semantics that can check if a system exhibits some behavior under *all* relative clock speeds.

Structure of the paper. In Section 2, we introduce our distributed automaton model with independently evolving clocks, and define its existential and universal semantics. Section 3 extends the regions of a timed automaton to our distributed setting, allowing us to compute a finite automaton recognizing the existential semantics. Section 4 shows that checking emptiness of the universal semantics is undecidable. This result is sharpened towards bounded clock drifts in Section 5. Section 6 deals with the reactive semantics, and Section 7 identifies some directions for future work.

A full version of this paper is available [1].

2 Distributed Timed Automata

Preliminaries. For a set Σ, we let Σ^* and Σ^ω denote the set of finite and, respectively, infinite words over Σ. The empty word is denoted by ε. We set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. The concatenation of words $u \in \Sigma^*$ and $v \in \Sigma^\infty$ is denoted by $u \cdot v$. An *alphabet* is a non-empty finite set. Given an alphabet Σ, we denote by Σ_ε the set $\Sigma \cup \{\varepsilon\}$. The set of non-negative real numbers is denoted by $\mathbb{R}_{\geq 0}$. For $t \in \mathbb{R}_{\geq 0}$, $\lfloor t \rfloor$ and $\mathit{fract}(t)$ refer to the integral and, respectively, fractional part of t, i.e., $t = \lfloor t \rfloor + \mathit{fract}(t)$.

The set $\mathrm{Form}(\mathcal{Z})$ of *clock formulas* over a set of clocks $\mathcal{Z}$ is given by the grammar $\varphi ::= \mathrm{true} \mid \mathrm{false} \mid x \bowtie C \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$ where x is a clock from $\mathcal{Z}$, $\bowtie \in \{<, \leq, >, \geq, =\}$, and C ranges over $\mathbb{N}$. A *clock valuation* over $\mathcal{Z}$ is a mapping $\nu : \mathcal{Z} \to \mathbb{R}_{\geq 0}$. We say that ν satisfies $\varphi \in \mathrm{Form}(\mathcal{Z})$, written $\nu \models \varphi$, if φ evaluates to true using the values given by ν. For $R \subseteq \mathcal{Z}$, $\nu[R]$ denotes the clock valuation defined by $\nu[R](x) = 0$ if $x \in R$ and $\nu[R](x) = \nu(x)$, otherwise.

The model. Let us recall the fundamental notion of timed automata [2]. These will constitute the building blocks of our distributed timed automata. A *timed automaton* is a tuple $\mathcal{A} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F)$ where S is a finite set of *states*, Σ is the alphabet of *actions*, $\mathcal{Z}$ is a finite set of *clocks*, $\delta \subseteq S \times \Sigma_\varepsilon \times \mathrm{Form}(\mathcal{Z}) \times 2^{\mathcal{Z}} \times S$ is the finite set of *transitions*, $I : S \to \mathrm{Form}(\mathcal{Z})$ associates with each state an *invariant*, $\iota \in S$ is the *initial state*, and $F \subseteq S$ is the set of *final states*. We let $\mathit{Reset}(\mathcal{A}) = \{x \in \mathcal{Z} \mid$ there is $(s, a, \varphi, R, s') \in \delta$ such that $x \in R\}$ be the set of clocks that might be reset in $\mathcal{A}$. Without loss of generality, we will assume in this paper that $I(\iota)$ is satisfied by the clock valuation over $\mathcal{Z}$ that maps each clock to 0.

We will now extend the above definition to a distributed setting. First, we fix a non-empty finite set Proc of processes (unless otherwise stated). For a tuple t that is indexed by Proc, t_p refers to the projection of t onto $p \in \mathit{Proc}$.

Definition 1. *A* distributed timed automaton (DTA) *over the set of processes Proc is a structure $\mathcal{D} = ((\mathcal{A}_p)_{p \in \mathit{Proc}}, \pi)$ where the $\mathcal{A}_p = (S_p, \Sigma_p, \mathcal{Z}_p, \delta_p, I_p, \iota_p, F_p)$ are timed automata and π is a mapping from $\bigcup_{p \in \mathit{Proc}} \mathcal{Z}_p$ to Proc such that, for each $p \in \mathit{Proc}$, we have $\mathit{Reset}(\mathcal{A}_p) \subseteq \pi^{-1}(p) \subseteq \mathcal{Z}_p$.*

Note that $\mathcal{Z}_p$ is the set of clocks that might occur in the timed automaton $\mathcal{A}_p$, either as clock guard or reset. The same clock may occur in both $\mathcal{Z}_p$ and $\mathcal{Z}_q$, since it may be read as a guard in both processes. However, any clock evolves according to the time evolution of some particular process. This clock is then said to *belong* to that process,

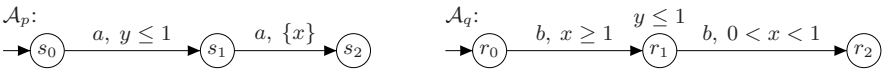


Fig. 1. A distributed timed automaton over $\{p, q\}$

and the *owner* map, π, formalizes this in the above definition. This will become more clear when we describe the formal semantics later in this section. Further, we assume that a clock can only be reset by the process it belongs to.

Example 2. Suppose $Proc = \{p, q\}$. Consider the DTA $\mathcal{D}$ as given by Figure 1. It consists of two timed automata, $\mathcal{A}_p$ and $\mathcal{A}_q$. In both automata, we suppose all states to be final. Moreover, the owner mapping π maps clock x to p and clock y to q. Note that $Reset(\mathcal{A}_p) = \{x\}$ and $Reset(\mathcal{A}_q) = \emptyset$. Before we define the semantics of $\mathcal{D}$ formally and in a slightly more general setting, let us give some intuitions on the behavior of $\mathcal{D}$. If both clocks are completely synchronized, i.e., they follow the same local clock rate, then our model corresponds to a standard network of timed automata [5]. For example, we might execute a within one time unit, and, after one time unit, execute b, ending up in the global state (s_1, r_1) and a clock valuation $\nu(x) = \nu(y) = 1$. If we now wanted to perform a further b, this should happen instantaneously. But this also requires a reset of x in the automaton $\mathcal{A}_p$ and, in particular, a time elapse greater than zero, violating the invariant at the local state r_1. Thus, the word $abab$ will not be in the semantics that we associate with $\mathcal{D}$ wrt. synchronized local-time evolution. Now suppose clock y runs slower than clock x. Then, having executed ab, we might safely execute a further a while resetting x and, then, let some time elapse without violating the invariant. Thus, $abab$ will be contained in the *existential* semantics, as there are local time evolutions that allow for the execution of this word. Observe that a and aa are the only sequences that can be executed no matter what the relative time speeds are: the guard $y \leq 1$ is always satisfied for a while. But we cannot guarantee that the guard $x \geq 1$ and the invariant $y \leq 1$ are satisfied at the same time, which prevents a word containing b from being in the *universal* semantics of $\mathcal{D}$.

The semantics. The semantics of a DTA depends on the (possibly dynamically changing) time rates at the processes. To model this, we assume that these rates depend on some absolute time, i.e., they are given by a tuple $\tau = (\tau_p)_{p \in Proc}$ of functions $\tau_p : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. Thus, each local time function maps every point in global time to some local time instant. Then, we require (justifiably) that these functions are continuous, strictly increasing, and divergent. Further, they satisfy $\tau_p(0) = 0$ for all $p \in Proc$. The set of all these tuples τ is denoted by $Rates$. We might consider τ as a mapping $\mathbb{R}_{\geq 0} \to (\mathbb{R}_{\geq 0})^{Proc}$ so that, for $t \in \mathbb{R}_{\geq 0}$, $\tau(t)$ denotes the tuple $(\tau_p(t))_{p \in Proc}$.

A distributed system can usually be described with an asynchronous product of automata. Indeed, the semantics of a DTA can be defined with such a product and a mapping that assigns any clock to its owner process: Let $\mathcal{D} = ((\mathcal{A}_p)_{p \in Proc}, \pi)$ with $\mathcal{A}_p = (S_p, \Sigma_p, \mathcal{Z}_p, \delta_p, I_p, \iota_p, F_p)$ be some DTA. We assign to $\mathcal{D}$ the asynchronous product $\mathcal{B}_\mathcal{D} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ as one might expect: We set $S = \prod_{p \in Proc} S_p$, $\Sigma = \bigcup_{p \in Proc} \Sigma_p$, $\mathcal{Z} = \bigcup_{p \in Proc} \mathcal{Z}_p$, $\iota = (\iota_p)_{p \in Proc}$, and $F = \prod_{p \in Proc} F_p$. Moreover, for any $s \in S$, we let $I(s) = \bigwedge_{p \in Proc} I_p(s_p)$. Finally, for $s, s' \in S$, $a \in \Sigma_\varepsilon$,

$\varphi \in \mathrm{Form}(\mathcal{Z})$, and $R \subseteq \mathcal{Z}$, we let $(s, a, \varphi, R, s') \in \delta$ if there is $p \in \mathit{Proc}$ such that $(s_p, a, \varphi, R, s'_p) \in \delta_p$ and $s_q = s'_q$ for each $q \in \mathit{Proc} \setminus \{p\}$.

Actually, most variants of a shared-memory model and their semantics can be unified by considering one single state space. This motivates the following definition:

Definition 3. *A* timed automaton with independently evolving clocks (icTA) *over* Proc *is a tuple* $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ *where* $(S, \Sigma, \mathcal{Z}, \delta, I, \iota, F)$ *is a timed automaton and* $\pi : \mathcal{Z} \to \mathit{Proc}$ *maps each clock to a process.*

Thus, the structure $\mathcal{B}_{\mathcal{D}}$ that we assigned to a DTA $\mathcal{D}$ is an icTA. Most of the following definitions and results are based on this more general notion of a timed system and therefore automatically carry over to the special case of DTAs. We will now define a run of an icTA. Intuitively, this is done in the same spirit as a run of a timed automaton over a timed word except for one difference. The time evolution, though according to absolute time, is perceived by each process as its *local time* evolution. So let $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ be an icTA. Given a clock valuation ν over $\mathcal{Z}$ and a tuple $t \in \mathbb{R}^{\mathit{Proc}}$, we let the valuation $\nu + t$ be given by $(\nu + t)(x) = \nu(x) + t_{\pi(x)}$ for all $x \in \mathcal{Z}$. For $\tau \in \mathit{Rates}$, a τ-run of $\mathcal{B}$ is a sequence $(s_0, \nu_0) \xrightarrow{a_1, t_1} (s_1, \nu_1) \dots (s_{n-1}, \nu_{n-1}) \xrightarrow{a_n, t_n} (s_n, \nu_n)$ where $n \geq 0$, $s_i \in S$, $a_i \in \Sigma_\varepsilon$, and $(t_i)_{1 \leq i \leq n}$ is a non-decreasing sequence of values from $\mathbb{R}_{\geq 0}$. Further, $\nu_i : \mathcal{Z} \to \mathbb{R}_{\geq 0}$ with $\nu_0(x) = 0$ for all $x \in \mathcal{Z}$. Finally, for all $i \in \{1, \dots, n\}$, there are $\varphi_i \in \mathrm{Form}(\mathcal{Z})$ and $R_i \subseteq \mathcal{Z}$ such that the following hold: $(s_{i-1}, a_i, \varphi_i, R_i, s_i) \in \delta$, $\nu_{i-1} + \tau(t') - \tau(t_{i-1}) \models I(s_{i-1})$ for each $t' \in [t_{i-1}, t_i]$, $\nu_{i-1} + \tau(t_i) - \tau(t_{i-1}) \models \varphi_i$, $\nu_i = (\nu_{i-1} + \tau(t_i) - \tau(t_{i-1}))[R_i]$, and $\nu_i \models I(s_i)$. In that case, we write $(\mathcal{B}, \tau) : s_0 \xrightarrow{a_1 \cdot \dots \cdot a_n} s_n$ or also $(\mathcal{B}, \tau) : s_0 \xrightarrow{a_1 \cdot \dots \cdot a_i} s_i \xrightarrow{a_{i+1} \cdot \dots \cdot a_n} s_n$ to abstract from the time instances. The latter thus denotes that $\mathcal{B}$ can, reading w, go from s_0 via s_i to s_n, while respecting the local-time behavior τ.

Definition 4. *Let* $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ *be an* icTA *and* $\tau \in \mathit{Rates}$. *The* language *of* $\mathcal{B}$ *wrt.* τ, *denoted by* $L(\mathcal{B}, \tau)$, *is the set of words* $w \in \Sigma^*$ *such that* $(\mathcal{B}, \tau) : \iota \xrightarrow{w} s$ *for some* $s \in F$. *Moreover, we define* $L_\exists(\mathcal{B}) = \bigcup_{\tau \in \mathit{Rates}} L(\mathcal{B}, \tau)$ *to be the* existential semantics *and* $L_\forall(\mathcal{B}) = \bigcap_{\tau \in \mathit{Rates}} L(\mathcal{B}, \tau)$ *to be the* universal semantics *of* $\mathcal{B}$.

If $|\mathit{Proc}| = 1$, then an icTA $\mathcal{B}$ actually reduces to an ordinary timed automaton and we have $L_\forall(\mathcal{B}) = L(\mathcal{B}, \tau) = L_\exists(\mathcal{B})$ for any $\tau \in \mathit{Rates}$. Moreover, if $|\mathit{Proc}| > 1$ and $\tau \in \mathit{Rates}$ exhibits, for all $p \in \mathit{Proc}$, the same local time evolution, then $L(\mathcal{B}, \tau)$ is the language of $\mathcal{B}$ considered as an ordinary timed automaton.

Example 5. A sample icTA $\mathcal{B}$ over set of processes $\{p, q\}$ and $\Sigma = \{a, b, c\}$ is depicted in Figure 2. Assuming $\pi^{-1}(p) = \{x\}$ and $\pi^{-1}(q) = \{y\}$, we have $L_\forall(\mathcal{B}) = \{a, ab\}$, $L(\mathcal{B}, \mathrm{id}) = \{a, ab, b\}$, and $L_\exists(\mathcal{B}) = \{a, ab, b, c\}$ where id_p is the identity on $\mathbb{R}_{\geq 0}$ for all $p \in \mathit{Proc}$ (i.e., id models synchronization of any process with the absolute time).

It is worthwhile to observe that $L(\mathcal{B}, \tau)$ can, in general, have bizarre (non-regular) behavior, if τ is itself a "weird" function. This is one more reason to look at the existential and universal semantics. Let us quantify this with an example. Consider the simple icTA $\mathcal{B}$ over $\mathit{Proc} = \{p, q\}$ fom Figure 3, where $\Sigma = \{a, b\}$, $\pi^{-1}(p) = \{x\}$, and

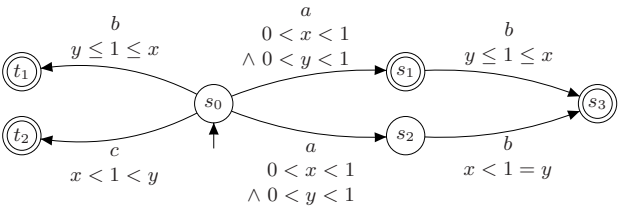


Fig. 2. An icTA $\mathcal{B}$ with independent clocks x and y

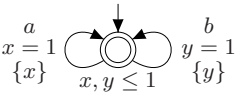


Fig. 3. A "weird" icTA

$\pi^{-1}(q) = \{y\}$. Further, let $\tau = (\mathrm{id}_p, \tau_q)$, where τ_q is any continuous, strictly increasing function such that $\tau_q(0) = 0$ and $\tau_q(n) = 2^n - 0.1$ for any $n \geq 1$. Then, $L(\mathcal{B}, \tau)$ is the set of finite prefixes of the infinite word $bab^2ab^4ab^8ab^{16}a\ldots$, which is not regular.

Finally, the semantics of a DTA is formally described in terms of its icTA.

Definition 6. *For a DTA $\mathcal{D}$ and $\tau \in Rates$, we set $L(\mathcal{D}, \tau) = L(\mathcal{B}_{\mathcal{D}}, \tau)$ to be the* language *of $\mathcal{D}$ wrt. τ, and we define $L_\exists(\mathcal{D}) = \bigcup_{\tau \in Rates} L(\mathcal{D}, \tau)$ as well as $L_\forall(\mathcal{D}) = \bigcap_{\tau \in Rates} L(\mathcal{D}, \tau)$ to obtain its existential and universal semantics, respectively.*

Example 7. For the DTA $\mathcal{D}$ from Figure 1, we can formalize what we had described intuitively: $L(\mathcal{D}, \mathrm{id}) = Pref(\{aab, aba, baa\})$, $L_\exists(\mathcal{D}) = Pref(\{aab, abab, baab\})$, and $L_\forall(\mathcal{D}) = Pref(\{aa\})$ where, for $L \subseteq \Sigma^*$, $Pref(L) = \{u \mid u, v \in \Sigma^*, u \cdot v \in L\}$.

3 Region Abstraction and the Existential Semantics

Given an icTA $\mathcal{B}$ and a set Bad of undesired behaviors, it is natural to ask if $\mathcal{B}$ is robust against the (unknown) relative clock speeds and faithfully avoids executing action sequences from Bad. This corresponds to checking if $L_\exists(\mathcal{B}) \cap Bad = \emptyset$. In this section, we show that this question is indeed decidable, given that Bad is a regular language. To this aim, we define a partitioning of clock valuations into finitely many equivalence classes and generalize the well-known region construction for timed automata [2].

Let $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ be an icTA. For a clock $x \in \mathcal{Z}$, let $C_x \in \mathbb{N}$ be the largest value the clock x is compared with in $\mathcal{B}$ (we assume that such a value exists). We say that two clock valuations ν and ν' over $\mathcal{Z}$ are *equivalent* if the following hold:

- for each $x \in \mathcal{Z}$, $\nu(x) > C_x$ iff $\nu'(x) > C_x$,
- for each $x \in \mathcal{Z}$, $\nu(x) \leq C_x$ implies both $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$ and ($fract(\nu(x)) = 0$ iff $fract(\nu'(x)) = 0$), and
- for each $p \in Proc$ and $x, y \in \pi^{-1}(p)$ such that $\nu(x) \leq C_x$ and $\nu(y) \leq C_y$, we have $fract(\nu(x)) \leq fract(\nu(y))$ iff $fract(\nu'(x)) \leq fract(\nu'(y))$.
  Note that this constraint only applies to clocks that belong to the same process.

An equivalence class of a clock valuation is called a *clock region* (of $\mathcal{B}$). For a valuation ν, $[\nu]$ denotes the clock region that contains ν. The set of clock regions of $\mathcal{B}$ is denoted by $Regions(\mathcal{B})$. Let γ and γ' be two clock regions, say with representatives ν and ν', respectively. We say that γ' is a *accessible* from γ, written $\gamma \preceq \gamma'$, if $\gamma' = \gamma$ or if there is $t \in (\mathbb{R}_{>0})^{Proc}$ such that $\nu' = \nu + t$. Note that $\preceq$ is a partial-order relation.

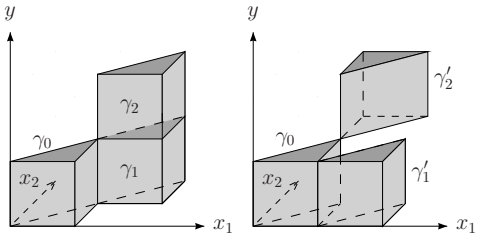


Fig. 4. Accessible and non-accessible regions

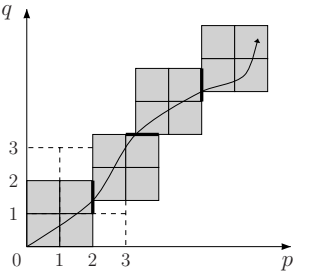


Fig. 5. $dir(\tau) = 010\ldots$

The *successor* relation, written $\gamma \prec\!\!\!\cdot\ \gamma'$, is as usual defined by $\gamma \prec \gamma'$ and $\gamma'' = \gamma$ or $\gamma'' = \gamma'$ for all clock regions γ'' with $\gamma \preceq \gamma'' \preceq \gamma'$.

Example 8. The accessible-regions relation is illustrated in Figure 4. Suppose we deal with two processes, one owning clocks x_1 and x_2, the other owning a single clock y. Suppose furthermore that, in the icTA at hand, all clocks are compared to the constant 2. Consider the prisms $\gamma_0, \gamma_1, \gamma_2, \gamma_1', \gamma_2'$, each representing a non-border clock region, which are given by the clock constraints $\gamma_0 = (0 < x_2 < x_1 < 1) \wedge (0 < y < 1)$, $\gamma_1' = (0 < x_2 < x_1 - 1 < 1) \wedge (0 < y < 1)$, $\gamma_1 = (1 < x_2 < x_1 < 2) \wedge (0 < y < 1)$, $\gamma_2' = (1 < x_1 < x_2 < 2) \wedge (1 < y < 2)$, and $\gamma_2 = (1 < x_2 < x_1 < 2) \wedge (1 < y < 2)$. We have $\gamma_0 \preceq \gamma_1 \preceq \gamma_2$. However, $\gamma_0 \not\preceq \gamma_1'$ and $\gamma_0 \not\preceq \gamma_2'$.

Let $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ be an icTA over *Proc*. We associate with $\mathcal{B}$ a non-deterministic finite automaton $\mathcal{R}_\mathcal{B} = (S', \Sigma, \delta', \iota', F')$, called the *region automaton* of $\mathcal{B}$, which is defined as follows: $S' = S \times \mathit{Regions}(\mathcal{B})$, $\iota' = (\iota, [\nu])$ where $\nu(x) = 0$ for all $x \in \mathcal{Z}$, $F' = F \times \mathit{Regions}(\mathcal{B})$, and for $a \in \Sigma_\varepsilon$, $s, s' \in S$, and $\gamma, \gamma' \in \mathit{Regions}(\mathcal{B})$, δ' contains $((s, \gamma), a, (s', \gamma'))$ if

- $a = \varepsilon$, $s = s'$, $\gamma \prec\!\!\!\cdot\ \gamma'$, and $\nu' \models I(s)$ for some $\nu' \in \gamma'$
  (we then call $((s, \gamma), a, (s', \gamma'))$ a *time-elapse transition*), or
- there are $\nu \in \gamma$ and $(s, a, \varphi, R, s') \in \delta$ such that $\nu \models \varphi \wedge I(s)$, $\nu[R] \models I(s')$, and $\nu[R] \in \gamma'$ (we then call $((s, \gamma), a, (s', \gamma'))$ a *discrete transition*).

A part of the region automaton for the icTA from Figure 2 is shown in Figure 10.

Indeed, the language $L(\mathcal{R}_\mathcal{B})$ of the non-deterministic finite automaton $\mathcal{R}_\mathcal{B}$, which is defined as usual, coincides with the existential semantics of $\mathcal{B}$:

Lemma 9. *Let $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ be an* icTA *and let C be the largest constant a clock is compared with in $\mathcal{B}$. Then, the number of states of $\mathcal{R}_\mathcal{B}$ is bounded by $|S| \cdot (2\,C + 2)^{|\mathcal{Z}|} \cdot |\mathcal{Z}|!$ and we have $L(\mathcal{R}_\mathcal{B}) = L_\exists(\mathcal{B})$.*

Thus, we solved the verification problem stated at the beginning of this section:

Theorem 10. *Model checking* icTAs *wrt. regular negative specifications is decidable.*

4 The Universal Semantics

While the existential semantics allows us to verify negative specifications, the universal semantics is natural when we want to check if our system has some good behavior. By good we mean a behavior that is robust against clock variations. Unfortunately, this problem is undecidable. This is shown for icTAs first and then will be extended to DTAs. Moreover, it turns out to be undecidable if, for a positive specification $Good$ containing the behaviors that a system *must* exhibit and an icTA $\mathcal{B}$, we have $Good \subseteq L_\forall(\mathcal{B})$.

Theorem 11. *The following problem is undecidable if* $|Proc| \geq 2$*: Given an* icTA $\mathcal{B}$ *over* $Proc$*, does* $L_\forall(\mathcal{B}) \neq \emptyset$ *hold?*

Proof. The proof is by reduction from Post's correspondence problem (PCP). An instance $Inst$ of the PCP consists of an alphabet A and two morphisms f and g from A^+ to $\{0,1\}^+$. A solution of $Inst$ is a word $w \in A^+$ such that $f(w) = g(w)$.

Suppose $Proc = \{p, q\}$ and let $\tau \in Rates$. One may associate with τ two sequences $t\text{-}dir(\tau) = t_1 t_2 \ldots \in (\mathbb{R}_{\geq 0})^\omega$ of time instances and $dir(\tau) = d_1 d_2 \ldots \in \{0,1,2\}^\omega$ of *directions* as follows: for $i \geq 1$, we let first (assuming $t_0 = 0$) $t_i = \min\{t > t_{i-1} \mid \tau_r(t) - \tau_r(t_{i-1}) = 2 \text{ for some } r \in Proc\}$. With this, we set

$$d_i = \begin{cases} 0 & \text{if } \tau_p(t_i) - \tau_p(t_{i-1}) = 2 \text{ and } 1 < \tau_q(t_i) - \tau_q(t_{i-1}) < 2 \\ 1 & \text{if } \tau_q(t_i) - \tau_q(t_{i-1}) = 2 \text{ and } 1 < \tau_p(t_i) - \tau_p(t_{i-1}) < 2 \\ 2 & \text{otherwise} \end{cases}$$

The construction of $dir(\tau)$ is illustrated in Figure 5. The idea is to allow the shape of the relative time-rate function (from τ) to encode a word in $\{0,1,2\}^\omega$. We do this using 2×2-square regions, each consisting of 4 sub-squares as shown. If the rate function leaves this region by the upper boundary or right boundary of the right-upper sub-square, then we write 1 or 0, respectively. If it leaves by any other boundary or by end-points of any sub-square, then we write 2. A new square region is started at the point where the rate function left the old one. Thus, the direction sequences partition the space of time rates.

Roughly speaking, a word is accepted universally by an icTA iff it is accepted for all directions. Our trick will be to define an icTA such that, the PCP instance has a solution w iff the word wb is accepted by the icTA for all directions. Thus, if there is no solution to the PCP, there will be some direction sequence (respectively, local time rates) for which the icTA does not accept.

Let an instance $Inst$ of the PCP be given by an alphabet $A = \{a_1, \ldots, a_k\}$ with $k \geq 1$ and two corresponding morphisms f and g. We will construct an icTA $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ over the set of processes $Proc = \{p, q\}$ and $\Sigma = \{a_1, \ldots, a_k, b\}$ such that $L_\forall(\mathcal{B}) = \{wb \mid w \in A^+ \text{ and } f(w) = g(w)\}$. First, let $\mathcal{Z} = \{x, y\}$ with $\pi(x) = p$ and $\pi(y) = q$. For $d \in \{0,1,2\}$, we set

$$guard(d) = \begin{cases} x = 2 \ \wedge \ 1 < y < 2 & \text{if } d = 0 \\ y = 2 \ \wedge \ 1 < x < 2 & \text{if } d = 1 \\ ((x \leq 1 \ \vee \ x = 2) \ \wedge y = 2) \ \vee \ (y \leq 1 \ \wedge \ x = 2) & \text{if } d = 2 \end{cases}$$

Moreover, let $\overline{guard}(d) = \bigvee_{d' \in \{0,1,2\} \setminus \{d\}} guard(d')$.

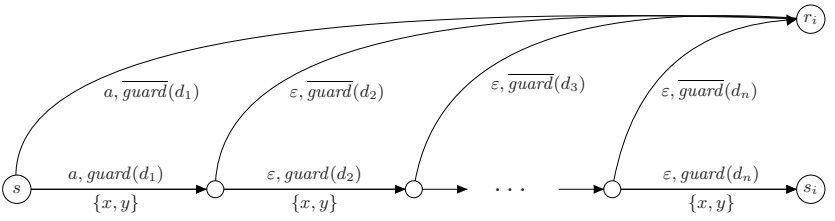


Fig. 6. Transition macro

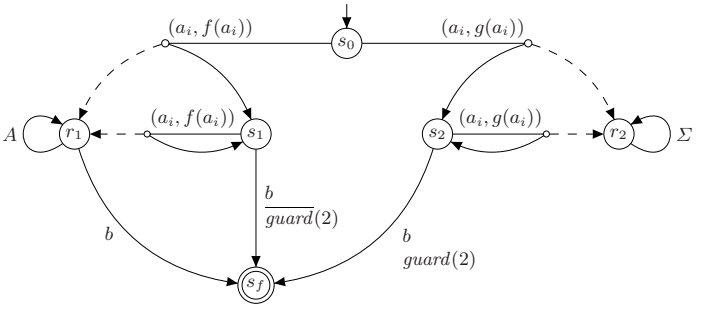
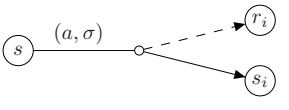


Fig. 7. Encoding of PCP

The final encoding of the given PCP instance in terms of the icTA is given by Figure 7. Hereby, given $a \in A$ and $\sigma = d_1 \dots d_n \in \{0, 1, 2\}^+$ (with $d_j \in \{0, 1, 2\}$ for any $j \in \{1, \dots, n\}$), a transition of the form



will actually stand for the sequence of transitions that is depicted in Figure 6, say, with intermediate states $s_{(i,a,\tau,1)}, \dots, s_{(i,a,\tau,n-1)}$.

Example 12. Consider the PCP instance *Inst* given by $A = \{a_1, a_2\}$, $f(a_1) = 101$, $g(a_1) = 1$, $f(a_2) = 1$, $g(a_2) = 01110$ with the obvious solution $w = a_1a_2a_1$. One can check that $a_1a_2a_1b \in L_\forall(\mathcal{B})$. This is illustrated in Figure 8. In the tree depicted, any path corresponds to a finite prefix (of length $|w| + 1$) of some sequence of directions. The edges are labeled by this sequence, where a left-edge is 0, downward is 2 and right-edge is 1. Thus, intuitively, a word wb is in the universal language iff all paths of the tree correspond to accepting runs in $\mathcal{B}$. Now, lets verify that the word wb is accepted by $\mathcal{B}$. If clock rate τ is such that $dir(\tau) \in f(w) \cdot d \cdot \{0, 1, 2\}^\omega$ with $d \in \{0, 1\}$, then the accepting run of $\mathcal{B}$ is the path shown in the left figure, which assigns states s_1 to nodes of the tree and finishes at s_f. If $d = 2$, then the accepting run of $\mathcal{B}$ is the path in the figure on right, which assigns states s_2 appropriately, crucially using the fact that $f(w) = g(w)$, and finally ends at s_f. If the clock rate τ has $dir(\tau)$ different from above

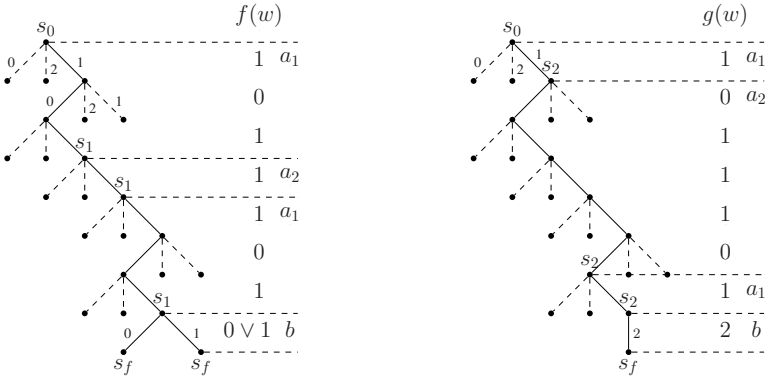


Fig. 8. The tree generated by $w = a_1a_2a_1b$ with respect to f and g

case, it is easy to see that there is an accepting run in which $\mathcal{B}$ reaches state s_f by passing through state r_1.

Let us show that our reduction is indeed correct. In the following, let $\leq$ denote the usual prefix relation on words.

Claim 13. For $\tau \in \mathit{Rates}$ and $w \in A^+$, the following hold:

(1) $f(w) \leq \mathit{dir}(\tau)$ iff $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} s_1$
(2) $g(w) \leq \mathit{dir}(\tau)$ iff $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} s_2$
(3) $f(w) \not\leq \mathit{dir}(\tau)$ iff $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} r_1$

With Claim 13, whose proof can be found in the full version [1], we can now show both directions of the correctness of the construction of $\mathcal{B}$.

Let $\tau \in \mathit{Rates}$ and suppose $f(w) = g(w)$. We distinguish three cases: If $\mathit{dir}(\tau) \in f(w) \cdot \{0,1\} \cdot \{0,1,2\}^\omega$, then, by (1), $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} s_1 \xrightarrow{b} s_f$. If $\mathit{dir}(\tau) \in f(w) \cdot 2 \cdot \{0,1,2\}^\omega$, then $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} s_2 \xrightarrow{b} s_f$. This follows from (2), since $g(w) = f(w)$. If $f(w) \not\leq \mathit{dir}(\tau)$, then, by (3), $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} r_1 \xrightarrow{b} s_f$. Hence, $wb \in L_\forall(\mathcal{B})$.

Let $w \in A^+$ and suppose $wb \in L_\forall(\mathcal{B})$. Pick $\tau \in \mathit{Rates}$ such that $\mathit{dir}(\tau) \in f(w) \cdot 2 \cdot \{0,1,2\}^\omega$. As $f(w) \leq \mathit{dir}(\tau)$, we have, by (3), $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} r_1$ and $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} s_1 \not\xrightarrow{b}$. Thus, we must have $(\mathcal{B}, \tau) : s_0 \xrightarrow{w} s_2 \xrightarrow{b} s_f$. Hence, by (2), $g(w) \cdot 2 \leq \mathit{dir}(\tau)$. As $f(w), g(w) \in \{0,1\}^*$, we have both $f(w) \cdot 2 \leq \mathit{dir}(\tau)$ and $g(w) \cdot 2 \leq \mathit{dir}(\tau)$, which implies $f(w) = g(w)$. □

Our result can be strengthened and extended to the distributed setting as follows:

Theorem 14. *Suppose $|\mathit{Proc}| \geq 2$. For DTAs $\mathcal{D}$ over Proc, the emptiness of $L_\forall(\mathcal{D})$ is undecidable.*

Proof. We fix $\mathit{Proc} = \{p, q\}$ and the clock distribution $\mathcal{Z}_p = \{x\}$ and $\mathcal{Z}_q = \{y\}$. Each process will be a copy of the automaton that is depicted in Figure 7, except for one

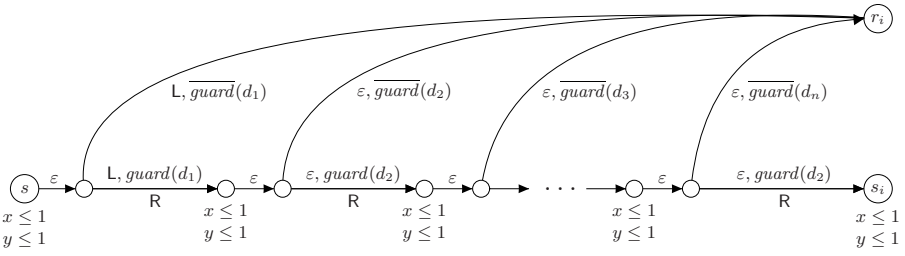


Fig. 9. Transition macro for the distributed setting

difference: for process p, the transition macro from Figure 6 is replaced with that from Figure 9 where L is the letter $a \in A$ and R is the singleton set $\{x\}$; for process q, we use the same new macro, but now we have $\mathsf{L} = \varepsilon$ and $\mathsf{R} = \{\mathrm{y}\}$.

To see how this works, we will just point out the difficulties and why the additional states with invariants and ε-transitions fix them. In the transition macro in Figure 6, clocks x and y belonging to different processes are reset at the same time. So, here we have two copies of the same automaton doing the same simulation but reset x in the automaton for process p, and y in the other. But this is not enough, since in the truly distributed setting, we cannot ensure that the clock resets are in sync. This might allow one process to wait while the other has reset its clock and thereby enable (wrong) transitions to state r_i, thus allowing the two automata copies to differ in the simulation. To ensure that the same path is followed, we split each state (except s_i and r_i) into two. The invariant on the first part then ensures that, before the next transition is enabled by the guard (which happens in the second part), both have been reset.

Let us examine this in more detail. Being in two identical copies of a state with an outgoing ε-transition, the ε-transitions might indeed be taken asynchronously by p and q. However, the following transitions will be performed synchronously by both processes. Assume first that p follows a transition of the form $(s_p, a, guard(d), \{x\}, s'_p)$ before process q moves. As $guard(d)$, where $d \in \{0, 1\}$, is satisfied when p goes to s'_p, the value of both clocks exceeds 1. But as x is reset at the same time whereas y is not, the invariant associated with s'_p is violated, which is a contradiction. Thus, q has to take the corresponding transition, which is of the form $(s_q, a, guard(d), \{y\}, s'_q)$, simultaneously. This explains why we use 2×2-squares as in Figure 5 and corresponding guards. In DTAs, they allow us to check when one clock has been reset and other has not. Now consider the case where p performs a transition of the form $(s_p, a, \overline{guard}(d), \emptyset, s'_p)$. When p executes its transition, at least one clock has reached the value 2. As this clock cannot be reset anymore, q is obliged to follow instantaneously the corresponding transition of the form $(s_q, a, \overline{guard}(d), \emptyset, s'_q)$, to reach a final state. □

Along the lines of the proofs of Theorems 11 and 14, we can show the following theorem, from which we derive the subsequent negative result (see [1] for details):

Theorem 15. *Suppose that $|Proc| \geq 2$. For DTAs $\mathcal{D}$ over Proc, it is undecidable if $L_\forall(\mathcal{D}) = \Sigma^*$ (where Σ is the set of actions of $\mathcal{B}_\mathcal{D}$).*

Theorem 16. *Model checking DTAs over at least two processes against regular positive specifications is undecidable.*

5 Playing with Local Time Rates

We have shown that it is undecidable to check if there is some word that is accepted under all clock rates by a given icTA $\mathcal{B}$. It is natural to ask if it is possible to restrict the independence of local time rates in some way to get decidability. For instance, we could insist that the ratio or the difference of local times in different processes must always be bounded. Unfortunately, this does not help. In fact, it turns out that our proof in Section 4 can be used to show that both these restrictions are already undecidable.

Let us formalize this. We will restrict to two processes, $Proc = \{p, q\}$. We note however that the following definitions can be easily generalized to more processes. For a rational number $k \geq 1$, we define $Rates_{\text{rat}}(k) = \{\tau = (\tau_p, \tau_q) \in Rates \mid \frac{1}{k} \leq \frac{\tau_p(t)}{\tau_q(t)} \leq k \text{ for all } t \in \mathbb{R}_{>0}\}$. This is the set of all rate-function tuples such that the ratio of the local times in the two processes are always bounded by fixed rationals. Further, for a rational number $\ell \geq 0$, $Rates_{\text{dif}}(\ell) = \{\tau = (\tau_p, \tau_q) \in Rates \mid |\tau_p(t) - \tau_q(t)| \leq \ell \text{ for all } t \in \mathbb{R}_{\geq 0}\}$. These are the rate function tuples for which the difference between the local times in the two processes are bounded by some constant. Accordingly, for an icTA or a DTA $\mathcal{B}$, we define $L_\forall^{\text{rat},k}(\mathcal{B}) = \bigcap_{\tau \in Rates_{\text{rat}}(k)} L(\mathcal{B}, \tau)$ and $L_\forall^{\text{dif},\ell}(\mathcal{B}) = \bigcap_{\tau \in Rates_{\text{dif}}(\ell)} L(\mathcal{B}, \tau)$.

Theorem 17. *For* icTAs *or DTAs $\mathcal{B}$ over $Proc = \{p, q\}$,*

1. *the emptiness of $L_\forall^{\text{rat},1}(\mathcal{B}) = L_\forall^{\text{dif},0}(\mathcal{B})$ is decidable.*
2. *the emptiness of $L_\forall^{\text{rat},k}(\mathcal{B})$ is undecidable for every rational $k > 1$.*
3. *the emptiness of $L_\forall^{\text{dif},\ell}(\mathcal{B})$ is undecidable for every rational $\ell > 0$.*

To prove the theorem, we need the following lemma.

Lemma 18. *Let $k > 1$, $\ell > 0$ be some fixed rationals. For all $\sigma \in \{0, 1, 2\}^*$, there exists $\tau \in Rates_{\text{rat}}(k) \cap Rates_{\text{dif}}(\ell)$ such that σ is a prefix of $dir(\tau)$.*

Proof. Let $\sigma = d_1 d_2 \dots d_n \in \{0, 1, 2\}^*$ be of length n. We define τ (in terms of $n + 1$ points) as follows: τ_p is the piecewise linear function with $\tau_p(2i) = x_i$ for $i \in \{0, \dots, n\}$ and $\tau_p(2n + t) = x_n + t$ for all $t \in \mathbb{R}_{\geq 0}$. Similarly, τ_q is defined as the piecewise linear function with $\tau_q(2i) = y_i$ for $i = \{0, \dots, n\}$ and $\tau_q(2n + t) = y_n + t$ for $t \in \mathbb{R}_{\geq 0}$. The points (x_i, y_i) are defined by $x_0 = y_0 = 0$ and, for $i \in \{1, \dots, n\}$, $x_i = 2i - \alpha|d_1 \dots d_i|_1$ and $y_i = 2i - \alpha|d_1 \dots d_i|_0$ ($|\sigma'|_d$ denoting the number of occurrences of d in σ'), where α is a rational parameter to be fixed.

With the above definition, we observe that, for all i, we have $|x_i - y_i| \leq i\alpha$, and, for $i > 0$, we have $1 - \frac{\alpha}{2} \leq \frac{x_i}{y_i} \leq \frac{1}{1-\alpha/2}$. Thus, by choosing $\alpha = \min\{\frac{\ell}{n}, 2(1 - \frac{1}{k})\}$, we can check that $\tau \in Rates_{\text{rat}}(k) \cap Rates_{\text{dif}}(\ell)$. Also it is easy to see that $dir(\tau) = \sigma \cdot 2^\omega$, which proves the lemma. □

Now, we can prove Theorem 17. For $k = 1$ or $\ell = 0$, the sets $Rates_{\text{rat}}(k)$ and $Rates_{\text{dif}}(\ell)$ consist of exactly the tuples in which time evolves at the same rate in both processes. Thus the sets are identical and correspond to an ordinary timed automaton so that emptiness is decidable.

Now, let $k > 1$ and $\ell > 0$. Given a PCP instance as before, we again consider the icTA (or DTA) $\mathcal{B}$ from Section 4. We want to show that $w \in A^+$ is solution iff $wb \in L_\forall(\mathcal{B}) = L_\forall^{\text{rat},k}(\mathcal{B}) = L_\forall^{\text{dif},\ell}(\mathcal{B})$. One direction is trivial. If, for $w \in A^+$, we have $f(w) = g(w)$, then $wb \in L_\forall(\mathcal{B})$, and this implies that $wb \in L_\forall^{\text{rat},k}(\mathcal{B})$ and $wb \in L_\forall^{\text{dif},\ell}(\mathcal{B})$. On the other hand, if $wb \in L_\forall^{\text{rat},k}(\mathcal{B})$ or $wb \in L_\forall^{\text{dif},\ell}(\mathcal{B})$, then, by Lemma 18, we pick $\tau \in Rates_{\text{rat}}(k) \cap Rates_{\text{dif}}(\ell)$ such that $dir(\tau) = f(w) \cdot 2 \cdot 2^\omega$, and the remaining part of the proof follows as before.

6 The Reactive Semantics

The universal semantics described in the previous section is a possible way to implement positive specifications, i.e, to make sure that our system must satisfy some behavior irrespective of the time/clock evolution. Unfortunately, since emptiness is undecidable even for bounded restrictions, it is not of any practical use. We would indeed like a semantics that describes only regular behaviors.

There is another subtle point for looking for other semantics. When we want to check if the system satisfies a positive specification, we would like to be able to design a controller which can actually do this. For this, the semantics has to be "reactive" in some sense. The universal semantics fails in this, in the sense that, to choose a correct run in the system, we might need to know the future time rates.

In this section, we introduce a new game-like semantics that solves both the above mentioned worries. It is regular and it is "reactive". Formally, we will describe it using an alternating automaton, which is based on the region automaton introduced in Section 3. Intuitively, *time-elapse* transitions are controlled by the environment whereas *discrete* transitions are controlled by the system that aims at exhibiting some behavior. This *game* is not *turn-based* because the system should be able to execute several discrete transitions while staying in the same region. After moving from some region to a successor region, the environment hands over the control to the system so that the system always has a chance to execute some discrete transition. On the other hand, after executing some discrete transition, the system may either keep the control or hand it over to the environment.

As suggested, our reactive semantics will be described by alternating automata. Since icTAs or DTAs have ε-transitions, we define an *alternating automaton with ε-transitions* (ε-AA) as a tuple $\mathcal{A} = (S, \Sigma, \delta, \iota, F)$ where S is a finite set of *states*, $\iota \in S$ is the *initial state*, $F \subseteq S$ is the set of *final states*, and $\delta : S \times \Sigma_\varepsilon \to \mathbb{B}^+(S)$ is the *alternating transition function*. Here, $\mathbb{B}^+(S)$ denotes positive boolean combinations of states from S.

As usual, a run of an ε-AA will be a (doubly) labeled finite tree. We assume the reader to be familiar with the notion of trees and only mention that we deal with structures (V, σ, μ) where V is the finite set of nodes with a distinguished root, and both σ and μ are node-labeling functions. Given a node $u \in V$, the set of children of u is denoted

$\mathrm{children}(u)$. Let $w = a_1 \dots a_{|w|} \in \Sigma^*$ be a finite word. A run of $\mathcal{A}$ on w is a doubly labeled finite tree $\rho = (V, \sigma, \mu)$ where $\sigma : V \to S$ is the *state-labeling* function and $\mu : V \to \{0, \dots, |w|\}$ is the *position-labeling* function such that, for each node $u \in V$, the following hold:

- if u is the root, then $\sigma(u) = \iota$ and $\mu(u) = 0$ (we start in the initial state at the beginning of the word),
- if u is not a leaf (i.e., $\mathrm{children}(u) \neq \emptyset$), then we have
  - either $\mu(u') = \mu(u)$ for all $u' \in \mathrm{children}(u)$ and in this case $\{\sigma(u') \mid u' \in \mathrm{children}(u)\} \models \delta(\sigma(u), \varepsilon)$
  - or $\mu(u') = \mu(u) + 1 = i \leq n$ for all $u' \in \mathrm{children}(u)$ and in this case $\{\sigma(u') \mid u' \in \mathrm{children}(u)\} \models \delta(\sigma(u), a_i)$.

The run is accepting if all leaves are labeled with $F \times \{|w|\}$. The set of words from Σ^* that come with an accepting run is denoted by $L(\mathcal{A})$.

Lemma 19 (cf. [4]). *Given an ε-AA $\mathcal{A}$ with n states, one can construct a non-deterministic finite automaton with $2^{\mathcal{O}(n^2)}$ states that recognizes $L(\mathcal{A})$.*

Let $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ be an icTA over *Proc*. We associate with $\mathcal{B}$ an ε-AA $\mathcal{A}_\mathcal{B} = (S', \Sigma, \delta', \iota', F')$ as follows: First, let $S' = S \times \mathit{Regions}(\mathcal{B}) \times \{0, 1\}$. Intuitively, tag 0 is for *system positions* while tag 1 is for *environment positions* (recall that the environment controls how time elapses whereas the system wants to accept some word). Then, $\iota' = (\iota, [\nu], 0)$ where $\nu(x) = 0$ for each $x \in \mathcal{Z}$, and $F' = F \times \mathit{Regions}(\mathcal{B}) \times \{0, 1\}$. Finally, for $(s, \gamma) \in S \times \mathit{Regions}(\mathcal{B})$ and $a \in \Sigma_\varepsilon$, we let

$$\delta'((s,\gamma,1),a) = \text{False} \quad \text{if } a \neq \varepsilon \qquad \delta'((s,\gamma,1),\varepsilon) = \bigwedge\{(s,\gamma',0) \mid \gamma \prec \gamma'\}$$

$$\delta'((s,\gamma,0),a) = \begin{cases} \bigvee\{(s',\gamma',0) \mid (s,\gamma) \xrightarrow{a}_{\mathrm{d}} (s',\gamma')\} & \text{if } a \neq \varepsilon \text{ or } \gamma \text{ maximal} \\ (s,\gamma,1) \vee \bigvee\{(s',\gamma',0) \mid (s,\gamma) \xrightarrow{\varepsilon}_{\mathrm{d}} (s',\gamma')\} & \text{otherwise} \end{cases}$$

where $\xrightarrow{a|\varepsilon}_{\mathrm{d}}$ denotes a discrete transition of the region automaton $\mathcal{R}_\mathcal{B}$ (Section 3).

Definition 20. *For an* icTA $\mathcal{B}$*, let $L_{react}(\mathcal{B}) = L(\mathcal{A}_\mathcal{B})$ be the* reactive semantics *of $\mathcal{B}$. Moreover, for a DTA $\mathcal{D}$, $L_{react}(\mathcal{D}) = L_{react}(\mathcal{B}_\mathcal{D})$ is the* reactive semantics *of $\mathcal{D}$.*

Example 21. Consider the icTA $\mathcal{B}$ from Figure 2. A part of its ε-AA $\mathcal{A}_\mathcal{B}$ is shown in Figure 10. States with tag 0 are depicted as ovals and are existential (non-deterministic) states and states with tag 1 are depicted as rectangles and are universal states. We have, e.g., $\delta'(r_1, \varepsilon) = r_3 \wedge r_4 \wedge r_5$. Note, however, that a transition from an oval to a rectangles should actually be split into two transitions, which is omitted in the picture. For example, there is a state r_1' between r_0 and r_1 which resembles r_1 but is tagged 0. Similarly, there is another state r_2' between r_0 and r_2, and we have $\delta'(r_0, a) = r_1' \vee r_2'$.

The following theorem follows from Lemma 19:

Theorem 22. *Let $\mathcal{B} = (S, \Sigma, \mathcal{Z}, \delta, I, \iota, F, \pi)$ be an* icTA *and let n be the number of states of $\mathcal{R}_\mathcal{B}$ (which is bounded by $|S| \cdot (2\,C+2)^{|\mathcal{Z}|} \cdot |\mathcal{Z}|!$ where C is the largest constant a clock is compared with in $\mathcal{B}$). Then, $L_{react}(\mathcal{B})$ is regular and one can compute a non-deterministic finite automaton with $2^{\mathcal{O}(n^2)}$ states that recognizes $L_{react}(\mathcal{B})$.*

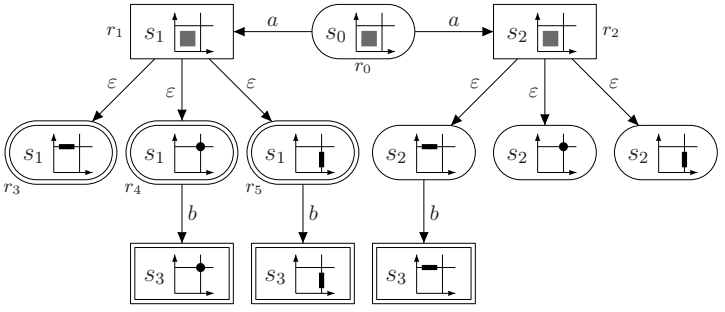


Fig. 10. Part of the region/alternating automaton for the icTA from Figure 2

The following inclusion property, whose proof can be found in [1], allows us to check an icTA for positive specifications. The subsequent proposition then establishes that inclusion actually forms a strict hierarchy of our semantics.

Proposition 23. *For any* icTA $\mathcal{B}$, $L_{react}(\mathcal{B}) \subseteq L_\forall(\mathcal{B})$.

Proposition 24. *Suppose that* $|Proc| \geq 2$. *There are some DTA* $\mathcal{D}$ *over* $Proc$ *and some* $\tau \in Rates$ *such that* $L_{react}(\mathcal{D}) \subsetneqq L_\forall(\mathcal{D}) \subsetneqq L(\mathcal{D}, \tau) \subsetneqq L_\exists(\mathcal{D})$.

Proof. Consider the icTA $\mathcal{B}$ from Figure 2. Recall that $L_{react}(\mathcal{B}) = \{a\}$, $L_\forall(\mathcal{B}) = \{a, ab\}$, $L(\mathcal{B}, \mathrm{id}) = \{a, ab, b\}$, and $L_\exists(\mathcal{B}) = \{a, ab, b, c\}$. As $\mathcal{B}$ does not employ any reset, we may view it as a DTA where $\mathcal{B}$ models a process owning clock x, and where a second process, owning clock y, does nothing, but is in a local accepting state. □

7 Future Work

We plan to investigate the expressive power of DTAs and, in particular, the *synthesis problem*: For which (global) specifications $Spec$ can we generate a DTA $\mathcal{D}$ (over some given system architecture) such that $L_{react}(\mathcal{D}) = Spec$? A similar synthesis problem has been studied in [8] in the framework of untimed distributed channel systems. There, additional messages are employed to achieve a given global behavior. In this context, it would be favorable to have partial-order based specification languages and a partial-order semantics for DTAs (see, for example, [11]).

References

1. Akshay, S., Bollig, B., Gastin, P., Mukund, M., Narayan Kumar, K.: Distributed timed automata with independently evolving clocks. Research Report LSV-08-19, ENS Cachan (2008)
2. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126(2), 183–235 (1994)
3. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998)

4. Birget, J.-C.: State-complexity of finite-state devices, state compressibility and incompressibility. Mathematical Systems Theory 26(3), 237–269 (1993)
5. Bouyer, P., Haddad, S., Reynier, P.-A.: Timed unfoldings for networks of timed automata. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218. Springer, Heidelberg (2006)
6. De Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robustness and implementability of timed automata. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 118–133. Springer, Heidelberg (2004)
7. Dima, C., Lanotte, R.: Distributed time-asynchronous automata. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711. Springer, Heidelberg (2007)
8. Genest, B.: On implementation of global concurrent systems with local asynchronous controllers. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 443–457. Springer, Heidelberg (2005)
9. Henzinger, T.A.: The theory of hybrid automata. In: Proc. of LICS 1996 (1996)
10. Larsen, K.G., Pettersson, P., Yi, W.: Compositional and symbolic model-checking of real-time systems. In: Proc. of RTSS 1995, p. 76. IEEE Computer Society, Los Alamitos (1995)
11. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. TCS 345(1), 27–59 (2005)
12. Puri, A.: Dynamical properties of timed automata. Discrete Event Dynamic Systems 10(1-2), 87–113 (2000)

A Context-Free Process as a Pushdown Automaton

J.C.M. Baeten, P.J.L. Cuijpers, and P.J.A. van Tilburg

Division of Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{j.c.m.baeten,p.j.l.cuijpers,p.j.a.v.tilburg}@tue.nl

Abstract. A well-known theorem in automata theory states that every context-free language is accepted by a pushdown automaton. We investigate this theorem in the setting of processes, using the rooted branching bisimulation and contrasimulation equivalences instead of language equivalence. In process theory, different from automata theory, interaction is explicit, so we realize a pushdown automaton as a regular process communicating with a stack.

1 Introduction

Automata and formal language theory have a place in every undergraduate computer science curriculum, as this provides students with a simple model of computation, and an understanding of computability. This simple model of computation does not include the notion of interaction, which is more and more important at a time when computers are always connected.

Adding interaction to automata theory leads to concurrency theory. The two models of computation are strongly related, and have much in common. Still, research into both models has progressed more or less independently. We are embarked on a program that studies similarities and differences between the two models, and that shows how concepts, notations, methods and techniques developed in one of the fields can be beneficial in the other field.

This paper studies, in a concurrency theoretic setting, the relation between the notion of a context-free process [4,18,9], and that of a pushdown automaton (i.e. a regular process that interacts with a stack) [17]. In order to obtain a full correspondence with automata theory, we extend the definition of context-free processes of [9] with deadlock ($\mathbf{0}$, as in [18]) and termination ($\mathbf{1}$, studied here for the first time). The goal of this paper, is to show how every context-free process can be translated into a pushdown automaton. The main difference with the work of [17], is that we do this while explicitly modeling the interaction between the regular process and the stack in this automaton. As it turns out, the addition of termination leads to additional expressivity of context-free processes, which in turn leads to a case distinction in the translation. Finally, the results in [17], show that the translation in the other direction is not always possible for context-free processes without termination, but as $\mathbf{1}$ gives us additional expressivity, it might hold in the new setting. However, as the translation in one direction is already not trivial, we leave the other direction as future work.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 98–113, 2008.

This paper is structured as follows. We first introduce our definitions of regular and context-free processes, and the associated equational theory, in Sects. 2 and 3, respectively. Then, in Sect. 4, we give the general structure of our translation, and study the different cases mentioned before as instances of this structure. We conclude the paper in Sect. 5, and give recommendations for future work.

2 Regular Processes

Before we introduce context-free processes, we first consider the notion of a regular process and its relation to regular languages in automata theory. We start with a definition of the notion of transition system from process theory. A finite transition system can be thought of as a non-deterministic finite automaton. In order to have a complete analogy, the transition systems we study have a subset of states marked as final states.

Definition 1 (Transition system). *A* transition system *M is a quintuple $(\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ where:*

1. *$\mathcal{S}$ is a set of states,*
2. *$\mathcal{A}$ is an alphabet,*
3. *$\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the set of* transitions *or* steps*,*
4. *$\uparrow \in \mathcal{S}$ is the initial state,*
5. *$\downarrow \subseteq \mathcal{S}$ is a set of final states.*

For $(s, a, t) \in \rightarrow$ we write $s \xrightarrow{a} t$. For $s \in \downarrow$ we write $s\downarrow$. A finite transition system *or* non-deterministic finite automaton *is a transition system of which the sets $\mathcal{S}$ and $\mathcal{A}$ are finite.*

In accordance with automata theory, where a *regular language* is a language equivalence class of a non-deterministic finite automaton, we define a *regular process* to be a bisimulation equivalence class [13] of a finite transition system. Contrary to automata theory, it is well-known that not every regular process has a *deterministic* finite transition system (i.e. a transition system for which the relation $\rightarrow$ is functional). The set of deterministic regular processes is a proper subset of the set of regular processes.

Next, consider the automata theoretic characterization of a regular language by means of a right-linear grammar. In process theory, a grammar is called a *recursive specification*: it is a set of recursive equations over a set of variables. A right-linear grammar then coincides with a recursive specification over a finite set of variables in the Minimal Algebra MA. (We use standard process algebra notation as propagated by [2,5]).

Definition 2. *The signature of* Minimal Algebra *MA is as follows:*

1. *There is a constant* **0***; this denotes inaction, a deadlock state; other names are δ or stop.*
2. *There is a constant* **1***; this denotes termination, a final state; other names are ε, skip or the empty process.*

3. *For each element of the alphabet $\mathcal{A}$ there is a unary operator $a._-$ called* action prefix*; a term $a.x$ will execute the elementary action a and then proceed as x.*
4. *There is a binary operator $+$ called* alternative composition*; a term $x+y$ will either execute x or execute y, a choice will be made between the alternatives.*

The constants **0** and **1** are needed to denote transition systems with a single state and no transitions. The constant **0** denotes a single state that is not a final state, while **1** denotes a single state that is also a final state.

Definition 3. *Let $\mathcal{V}$ be a set of* variables*. A* recursive specification *over $\mathcal{V}$ with initial variable $S \in \mathcal{V}$ is a set of equations of the form $X = t_X$, exactly one for each $X \in \mathcal{V}$, where each right-hand side t_X is a term over some signature, possibly containing elements of $\mathcal{V}$. A recursive specification is called finite, if $\mathcal{V}$ is finite.*

We find that a finite recursive specification over MA can be seen as a right-linear grammar. Now each finite transition system corresponds directly to a finite recursive specification over MA, using a variable for every state. To go from a term over MA to a transition system, we use *structural operational semantics* [1], with rules given in Table 1.

Table 1. Operational rules for MA and recursion ($a \in \mathcal{A}, X \in \mathcal{V}$)

$$\frac{}{\mathbf{1}\downarrow} \qquad \frac{}{a.x \xrightarrow{a} x}$$

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} \qquad \frac{x\downarrow}{x + y\downarrow} \qquad \frac{y\downarrow}{x + y\downarrow}$$

$$\frac{t_X \xrightarrow{a} x \quad X = t_X}{X \xrightarrow{a} x} \qquad \frac{t_X\downarrow \quad X = t_X}{X\downarrow}$$

3 Context-Free Processes

Considering the automata theoretic notion of a context-free grammar, we find a correspondence in process theory by taking a recursive specification over a finite set of variables, and over the *Sequential Algebra* SA, which is MA extended with sequential composition $\_\cdot\_$. We extend the operational rules of Table 1 with rules for sequential composition, in Table 2.

Now consider the following specification

$$S = \mathbf{1} + S \cdot a.\mathbf{1}.$$

Our first observation is that, by means of the operational rules, we derive an infinite transition system, which moreover is infinitely branching. All the states of this transition system are different in bisimulation semantics, and so this

Table 2. Operational rules for sequential composition ($a \in \mathcal{A}$)

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x\downarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'} \qquad \frac{x\downarrow \quad y\downarrow}{x \cdot y\downarrow}$$

is in fact an infinitely branching process. Our second observation is that this recursive specification has infinitely many different (non-bisimilar) solutions in the transition system model, since adding any non-terminating branch to the initial node will also give a solution. This is because the equation is *unguarded*, the right-hand side contains a variable that is not in the scope of an action-prefix operator, and also cannot be brought into such a form. So, if there are multiple solutions to a recursive specification, we have multiple processes that correspond to this specification. This is an undesired property.

These two observations are the reason to restrict to guarded recursive specifications only. It is well-known that a guarded recursive specification has a unique solution in the transition system model (see [7,6]). This restriction leads to the following definition.

Definition 4. *A* context-free process *is the bisimulation equivalence class of the transition system generated by a finite guarded recursive specification over Sequential Algebra SA.*

In this paper, we use equational reasoning to manipulate recursive specifications. The equational theory of SA is given in Table 3. Note that the axioms $x \cdot (y+z) = x \cdot y + x \cdot z$ and $x \cdot \mathbf{0} = \mathbf{0}$ do not hold in bisimulation semantics (in contrast to language equivalence). The given theory constitutes a sound and ground-complete axiomatization of the model of transition systems modulo bisimulation (see [6,5]). Furthermore, we often use the aforementioned principle, that guarded recursive specifications have unique solutions [6].

Table 3. Equational theory of SA ($a \in \mathcal{A}$)

$x + y$	$= y + x$	$x + \mathbf{0}$	$= x$
$(x + y) + z$	$= x + (y + z)$	$\mathbf{0} \cdot x$	$= \mathbf{0}$
$x + x$	$= x$	$\mathbf{1} \cdot x$	$= x$
$(x + y) \cdot z$	$= x \cdot z + y \cdot z$	$x \cdot \mathbf{1}$	$= x$
$(x \cdot y) \cdot z$	$= x \cdot (y \cdot z)$	$(a.x) \cdot y$	$= a.(x \cdot y)$

Using the axioms, any guarded recursive specification can be brought into *Greibach normal form* [4]:

$$X = \sum_{i \in I_X} a_i.\xi_i \ (+ \mathbf{1}).$$

In this form, every right-hand side of every equation consists of a number of summands, indexed by a finite set I_X (the empty sum is $\mathbf{0}$), each of which is $\mathbf{1}$, or of the form $a_i.\xi_i$, where ξ_i is the sequential composition of a number of variables (the empty sequence is $\mathbf{1}$). We define $\mathcal{I}$ as the multiset resulting of the union of all index sets. For a recursive specification in Greibach normal form, every state of the transition system is given by a sequence of variables. Note that we can take the index sets associated with the variables to be disjoint, so that we can define a function $V : \mathcal{I} \to \mathcal{V}$ that gives, for any index that occurs somewhere in the specification, the variable of the equation in which it occurs.

As an example, we consider the important context-free process *stack*. Suppose D is a finite data set, then we define the following actions in $\mathcal{A}$, for each $d \in D$:

- $?d$: push d onto the stack;
- $!d$: pop d from the stack.

Now the recursive specification is as follows:

$$S = \mathbf{1} + \sum_{d \in D} ?d.S \cdot !d.S.$$

In order to see that the above process indeed defines a stack, define processes S_σ, denoting the stack with contents $\sigma \in D^*$, as follows: the first equation for the empty stack, the second for any nonempty stack, with top d and tail σ:

$$S_\varepsilon = S, \qquad S_{d\sigma} = S \cdot !d.S_\sigma.$$

Then it is straightforward to derive the following equations:

$$S_\varepsilon = \mathbf{1} + \sum_{d \in D} ?d.S_d, \qquad S_{d\sigma} = !d.S_\sigma + \sum_{e \in D} ?e.S_{ed\sigma}.$$

We obtain the following specification for the stack in Greibach normal form:

$$S = \mathbf{1} + \sum_{d \in D} ?d.T_d \cdot S, \qquad T_d = !d.\mathbf{1} + \sum_{e \in D} ?e.T_e \cdot T_d.$$

Finally, we define the *forgetful stack*, which can forget a datum it has received when popped, as follows:

$$S = \mathbf{1} + \sum_{d \in D} ?d.S \cdot (\mathbf{1} + !d.S).$$

Due to the presence of $\mathbf{1}$, a context-free process may have unbounded branching [8] that we need to mimic with our pushdown automaton. One possible solution is to use forgetfulness of the stack to get this unbounded branching in our pushdown automaton, as we will show in the next section. Note that

when using a more restrictive notion of context-free processes we have bounded branching, and thus we don't need the forgetfulness property.

The above presented specifications are still meaningful when D is an infinite data set (see e.g. [15,14]), but does not represent a term in SA anymore. In this paper, we use infinite summation in some intermediate results, but the end results are finite. Note that the infinite sums also preserve the notion of congruence we are working with.

Now, consider the notion of a pushdown automaton. A pushdown automaton is just a finite automaton, but at every step it can push a number of elements onto a stack, or it can pop the top of the stack, and take this information into account in determining the next move. Thus, making the interaction explicit, a pushdown automaton is a regular process communicating with a stack.

In order to model the interaction between the regular process and the stack, we briefly introduce communication by synchronization. We introduce the *Communication Algebra* CA, which extends MA and SA with the *parallel composition* operator $\|$. Parallel processes can execute actions independently (called interleaving), or can synchronize by executing matching actions. In this paper, it is sufficient to use a particular communication function, that will only synchronize actions $!d$ and $?d$ (for the same $d \in D$). The result of such a synchronization is denoted $⁉d$. CA also contains the *encapsulation operator* $\partial_*(\_)$, which blocks actions $!d$ and $?d$, and the *abstraction operator* $\tau_*(\_)$ which turns all $⁉d$ actions into the internal action τ. We show the operational rules in Table 4.

Table 4. Operational rules for CA ($a \in \mathcal{A}$)

$$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y} \qquad \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'} \qquad \frac{x\!\downarrow \quad y\!\downarrow}{x \parallel y\!\downarrow}$$

$$\frac{x \xrightarrow{?d} x' \quad y \xrightarrow{!d} y'}{x \parallel y \xrightarrow{⁉d} x' \parallel y'} \qquad \frac{x \xrightarrow{!d} x' \quad y \xrightarrow{?d} y'}{x \parallel y \xrightarrow{⁉d} x' \parallel y'}$$

$$\frac{x \xrightarrow{a} x' \quad a \neq !d, ?d}{\partial_*(x) \xrightarrow{a} \partial_*(x')} \qquad \frac{x\!\downarrow}{\partial_*(x)\!\downarrow}$$

$$\frac{x \xrightarrow{⁉d} x'}{\tau_*(x) \xrightarrow{\tau} \tau_*(x')} \qquad \frac{x \xrightarrow{a} x' \quad a \neq ⁉d}{\tau_*(x) \xrightarrow{a} \tau_*(x')} \qquad \frac{x\!\downarrow}{\tau_*(x)\!\downarrow}$$

Our finite axiomatization of transition systems of CA modulo rooted branching bisimulation uses the auxiliary operators $\_ \between \_$ and $\_ \mid \_$ [7,16]. See Table 5 for the axioms and [5] for an explanation of these axioms.

The given equational theory is sound and ground-complete for the model of transition systems modulo rooted branching bisimulation [13]. This is the preferred model we use, but all our reasoning in the following takes place in the

Table 5. Equational theory of CA ($a \in \mathcal{A} \cup \{\tau\}$)

$x \parallel y$	$= x \between y + y \between x + x \mid y$	$a.(\tau.(x+y)+x)$	$= a.(x+y)$
$\mathbf{0} \between x$	$= \mathbf{0}$	$x \mid y$	$= y \mid x$
$\mathbf{1} \between x$	$= \mathbf{0}$	$x \parallel \mathbf{1}$	$= x$
$a.x \between y$	$= a.(x \parallel y)$	$\mathbf{1} \mid x + \mathbf{1}$	$= \mathbf{1}$
$(x+y) \between z$	$= x \between z + y \between z$	$(x \parallel y) \parallel z$	$= x \parallel (y \parallel z)$
$\mathbf{0} \mid x$	$= \mathbf{0}$	$(x \mid y) \mid z$	$= x \mid (y \mid z)$
$(x+y) \mid z$	$= x \mid z + y \mid z$	$(x \between y) \between z$	$= x \between (y \parallel z)$
$\mathbf{1} \mid \mathbf{1}$	$= \mathbf{1}$	$(x \mid y) \between z$	$= x \mid (y \between z)$
$a.x \mid \mathbf{1}$	$= \mathbf{0}$	$x \between \tau.y$	$= x \between y$
$!d.x \mid ?d.y$	$= ⁉d.(x \parallel y)$	$x \mid \tau.y$	$= \mathbf{0}$
$a.x \mid b.y$	$= \mathbf{0}$ *if* $\{a,b\} \neq \{!d,?d\}$		
$\partial_*(\mathbf{0})$	$= \mathbf{0}$	$\tau_*(\mathbf{0})$	$= \mathbf{0}$
$\partial_*(\mathbf{1})$	$= \mathbf{1}$	$\tau_*(\mathbf{1})$	$= \mathbf{1}$
$\partial_*(!d.x)$	$= \partial_*(?d.x) = \mathbf{0}$	$\tau_*(⁉d.x)$	$= \tau.\tau_*(x)$
$\partial_*(a.x)$	$= a.\partial_*(x)$ *if* $a \notin \{!d,?d\}$	$\tau_*(a.x)$	$= a.\tau_*(x)$ *if* $a \neq ⁉d$
$\partial_*(x+y)$	$= \partial_*(x) + \partial_*(y)$	$\tau_*(x+y)$	$= \tau_*(x) + \tau_*(y)$

equational theory, so is model-independent provided the models preserve validity of the axioms and unique solutions for guarded recursive specifications.

4 Pushdown Automata

The main goal of this paper, is to prove that every context-free process is equal to a regular process communicating with a stack. Thus, if P is any context-free process, then we want to find a regular process Q such that

$$P = \tau_*(\partial_*(Q \parallel S_\sigma)),$$

where S_σ is a state of a stack process. Without loss of generality, we assume in this section that P is given in Greibach normal form.

The first, intermediate, solution we present uses a potentially infinite data type D. If D is infinite, then the stack is not a context-free process. Also, we define Q in the syntax of Minimal Algebra, but it may have infinitely many different variables, so it may not be a regular process. Later, we specialize to cases where the data type is finite, and these problems do not occur. We do this by reducing the main solution using several assumptions, that categorize the possibilities for P into three classes: *opaque*, *bounded branching*, and *unrestricted* specifications.

4.1 Intermediate Solution

The infinite data type D we use for our intermediate solution, consists of pairs. The first element of the pair is a variable of P. The second element is a multiset

over $\mathcal{I}$, i.e. a multiset over V, plus an indication of a termination option. So, $D = \mathcal{V} \times (\mathcal{I} \cup \{\mathbf{1}\} \to \mathbb{N})$.

For multisets A, B, we write $A(a) = n$ if the element a occurs n times in A, and we write $A \uplus B$ to denote union of multisets such that $(A \uplus B)(a) = A(a) + B(a)$. We use the subscript c in a process term $(p)_c$ to denote that p only occurs in the term if condition c holds. Finally, we call a variable *transparent* if its equation has an $\mathbf{1}$-summand. We denote the set of transparent variables of P with $\mathcal{V}^{+\mathbf{1}}$.

Now, we prove the main theorem by first stating the specification of our solution and introducing some formalisms, before giving the main proof. The proof will provide insight in how and why our solution works.

Theorem 1. *For every context-free process P there exists a process Q given by a recursive specification over MA such that $P = \tau_*(\partial_*(Q \parallel S_\sigma))$ for some state S_σ of the (partially) forgetful stack.*

Proof. Let E be a finite recursive specification of P in Greibach normal form. Now, let F be a recursive specification that contains the following equations for every variable $X \in \mathcal{V}$ of the specification E, $i \in I_X$ and multiset A over $\mathcal{I}$:

$$\hat{X}(i, A) = \mathrm{Push}(\xi_i, A),$$

with $\mathrm{Push}(\xi, A)$ recursively defined as

$$\mathrm{Push}(\mathbf{1}, A) = \mathrm{Ctrl}(A),$$

$$\mathrm{Push}(\xi' Y, A) = \begin{cases} !\langle Y, A\rangle.\mathrm{Push}(\xi', I_Y) & \text{if } Y \notin \mathcal{V}^{+\mathbf{1}}, \\ !\langle Y, A\rangle.\mathrm{Push}(\xi', I_Y \uplus A) & \text{if } Y \in \mathcal{V}^{+\mathbf{1}}. \end{cases}$$

where Y is a variable at the end of the original sequence and ξ' is the sequence that is left over when Y has been removed. So, $\mathrm{Push}(\xi, A)$ is defined backwards with respect to sequence ξ, necessary to preserve the correct structure on the stack while pushing.

In addition, let F also contain the following equations of a *partially forgetful stack* and a (regular) finite control.

$$S = \mathbf{1} + \sum_{\substack{\langle V, A\rangle \in D \\ V \notin \mathcal{V}^{+\mathbf{1}}}} ?\langle V, A\rangle.S \cdot !\langle V, A\rangle.S + \sum_{\substack{\langle V, A\rangle \in D \\ V \in \mathcal{V}^{+\mathbf{1}}}} ?\langle V, A\rangle.S \cdot (\mathbf{1} + !\langle V, A\rangle.S),$$

$$\mathrm{Ctrl}(A) = \sum_{i \in \mathcal{I}} \sum_{0 < l \le A(i)} a_i.\mathrm{Pop}(i, l)\ (+\,\mathbf{1})_{A(\mathbf{1}) \ge 1},$$

$$\mathrm{Pop}(i, l) = \begin{cases} \displaystyle\sum_{\substack{\langle V, A\rangle \in D \\ i \in I_V}} ?\langle V, A\rangle.\hat{V}(i, A) & \text{if } V(i) \notin \mathcal{V}^{+\mathbf{1}}, \\ \displaystyle\sum_{\substack{\langle V, A\rangle \in D \\ i \in I_V \wedge A(i) = l-1}} ?\langle V, A\rangle.\hat{V}(i, A) & \text{if } V(i) \in \mathcal{V}^{+\mathbf{1}}, \end{cases}$$

The process $\mathrm{Ctrl}(A)$ allows for a choice to be made among the possible enabled actions a_i, referred to by the indices in the multiset A. It can also terminate if the termination option $\mathbf{1}$ is present in A. Once an action has been chosen, Ctrl calls Pop with the index i of the action that was executed and the occurrence l of the variable belonging to that index, $V(i)$, on the stack that needs to be popped. Once that variable, say $V(i) = X$, has been popped, $\hat{X}(i, A)$ is executed to mimic the rest of the behavior when a_i has been executed, namely pushing ξ_i on the stack. Note that this means that A, the multiset of possible actions, always has to correspond with the contents of the partially forgetful stack.

Before we show how the above specification mimics the specification of P, we first study the structure of P itself more closely. In Greibach normal form, every state in P is labeled with a sequential composition of variables $X\xi$ (or in the trivial case, $\mathbf{1}$). Substituting the Greibach normal form of the leading variable X gives us the following:

$$X\xi = \Big(\sum_{i \in I_X} a_i.\xi_i \ (+\,\mathbf{1}) \Big) \cdot \xi = \sum_{i \in I_X} a_i.\xi_i \cdot \xi \ (+\,\xi).$$

Introducing a fresh variable $\overline{P}(\xi)$ for each possible sequence ξ, we obtain the following equivalent infinite recursive specification.

$$\overline{P}(\mathbf{1}) = \mathbf{1}, \qquad \overline{P}(X\xi) = \sum_{i \in I_X} a_i.\overline{P}(\xi_i\xi) \ (+\,\overline{P}(\xi)).$$

Note that this specification is still guarded, as the unfolding of the unguarded recursion will always terminate.

In order to link the sequences that make up the states of P to the contents of the stack in our specification F, we use two functions h and e. The function h determines, for a given sequence $X\xi$, the multiset that contains for each index $i \in \mathcal{I}$ the number of occurrences of the process variable $V(i)$ in a sequence that is reachable through termination of preceding variables. It also determines whether a termination is possible through the entire sequence.

$$h(\mathbf{1}) = \{\mathbf{1}\},$$
$$h(X\xi) = \begin{cases} I_X & \text{if } X \notin \mathcal{V}^{+\mathbf{1}}, \\ I_X \uplus h(\xi) & \text{if } X \in \mathcal{V}^{+\mathbf{1}}. \end{cases}$$

The function e, defined by $e(\mathbf{1}) = \mathbf{1}$ and $e(X\xi) = \langle X, h(\xi)\rangle e(\xi)$, then represents the actual contents of the stack.

Lemma 1. *Let $i \in \mathcal{I}$. Then $h(X\xi)(i) = h(\xi)(i)$ iff $i \notin I_X$.*

Having characterized the relationship between states of P and the partially forgetful stack of F, we define $Q = \mathrm{Ctrl}(h(X))$, where X is the initial variable of E, and we continue to prove $P = \tau_*(\partial_*(Q \parallel S_{e(X)})) = \big[Q \parallel S_{e(X)}\big]_*$.[1] More precisely, we will prove for any sequence of variables ξ, that

[1] From here on, $[p]_*$ is used as a shorthand notation for $\tau_*(\partial_*(p))$.

$$\overline{P}(\xi) = \left[\mathrm{Ctrl}(h(\xi)) \parallel S_{e(\xi)}\right]_* .$$

1. If $\xi = \mathbf{1}$, then $\overline{P}(\mathbf{1}) = \left[\mathrm{Ctrl}(h(\mathbf{1})) \parallel S_{e(\mathbf{1})}\right]_* = [\mathbf{1} \parallel S_{\mathbf{1}}]_* = \mathbf{1}$
2. If $\xi = X\xi'$, then there are two cases.
   (a) Assume that $X \notin \mathcal{V}^{+\mathbf{1}}$. First, apply the definition of $h(X\xi')$ and then the definition of $\mathrm{Ctrl}(I_X)$.

$$\begin{aligned}
\overline{P}(X\xi') &\stackrel{?}{=} \left[\mathrm{Ctrl}(h(X\xi')) \parallel S_{e(X\xi')}\right]_* \\
&= \left[\mathrm{Ctrl}(I_X) \parallel S_{e(X\xi')}\right]_* \\
&= \sum_{i\in\mathcal{I}} \sum_{0<l\le I_X(i)} a_i.\left[\mathrm{Pop}(i,l) \parallel S_{e(X\xi')}\right]_* \ (+\ \left[\mathbf{1} \parallel S_{e(X\xi')}\right]_*)_{I_X(\mathbf{1})\ge 1}
\end{aligned}$$

   Note that I_X is a set, so it follows that $I_X(i) = 1$ for $i \in I_X$ and $I_X(i) = 0$ for all $i \in \mathcal{I} - I_X$. Therefore, the first two summations can be written as $\sum_{i\in I_X}$ when we instantiate $l = 1$. Because it also follows that $I_X(\mathbf{1}) = 0$, we remove the conditional summand $\left[\mathbf{1} \parallel S_{e(X\xi')}\right]_*$.

$$= \sum_{i\in I_X} a_i.\left[\mathrm{Pop}(i,1) \parallel S_{e(X\xi')}\right]_*$$

   Unfold the definition of $S_{e(X\xi')}$ once, then perform the pop by applying the definitions of $\mathrm{Pop}(i,1)$ and $\hat{X}(i,h(\xi'))$.

$$\begin{aligned}
&= \sum_{i\in I_X} a_i.\tau.\left[\hat{X}(i,h(\xi')) \parallel S_{e(\xi')}\right]_* \\
&= \sum_{i\in I_X} a_i.\tau.\left[\mathrm{Push}(\xi_i,h(\xi')) \parallel S_{e(\xi')}\right]_*
\end{aligned}$$

   Finally, perform $|\xi_i|$ pushes by repeatedly applying the definitions of $\mathrm{Push}(\xi,A)$ and $S_{e(\xi)}$.

$$\begin{aligned}
&= \sum_{i\in I_X} a_i.\tau^{|\xi_i|+1}.\left[\mathrm{Ctrl}(h(\xi_i\xi')) \parallel S_{e(\xi_i\xi')}\right]_* \\
&= \sum_{i\in I_X} a_i.\left[\mathrm{Ctrl}(h(\xi_i\xi')) \parallel S_{e(\xi_i\xi')}\right]_* \\
&= \sum_{i\in I_X} a_i.\overline{P}(\xi_i\xi').
\end{aligned}$$

   (b) Assume that $X \in \mathcal{V}^{+\mathbf{1}}$. First, substitute the definition of $\mathrm{Ctrl}(h(X\xi'))$.

$$\begin{aligned}
\overline{P}(X\xi') &\stackrel{?}{=} \left[\mathrm{Ctrl}(h(X\xi')) \parallel S_{e(X\xi')}\right]_* \\
&= \sum_{i\in\mathcal{I}} \sum_{0<l\le h(X\xi')(i)} a_i.\left[\mathrm{Pop}(i,l) \parallel S_{e(X\xi')}\right]_*
\end{aligned}$$

Split off the case that will pop the top element of the stack, namely when $i \in I_X$ and $l = h(X\xi')(i)$. By the same argument as in the previous case, we can write the first two summations as $\sum_{i \in I_X}$.

$$
\begin{aligned}
&= \sum_{i \in I_X} a_i . \left[\mathrm{Pop}(i, h(X\xi')(i)) \parallel S_{e(X\xi')}\right]_* \\
&\quad + \sum_{i \in \mathcal{I}} \sum_{\substack{0 < l \leq h(X\xi')(i) \\ i \notin I_X \vee l \neq h(X\xi')(i)}} a_i . \left[\mathrm{Pop}(i, l) \parallel S_{e(X\xi')}\right]_* \\
&\quad (+ \left[\mathbf{1} \parallel S_{e(X\xi')}\right]_*)_{h(X\xi')(\mathbf{1}) \geq 1}
\end{aligned}
$$

Consider the first summation. If $i \in I_X$ and $l = h(X\xi')(i)$, then $h(\xi')(i) = l - 1$ by Lemma 1 and therefore by the definitions of $\mathrm{Pop}(i, l)$ and $S_{e(X\xi')}$:

$$
\begin{aligned}
&= \sum_{i \in I_X} a_i . \left[\sum_{\substack{\langle V, A' \rangle \in D \\ i \in I_v \wedge A'(i) = l - 1}} ?\langle V, A' \rangle . \hat{X}(i, A') \parallel S_{\langle X, h(\xi') \rangle e(\xi')} \right]_* \\
&\quad + \sum_{i \in \mathcal{I}} \sum_{\substack{0 < l \leq h(X\xi')(i) \\ i \notin I_X \vee l \neq h(X\xi')(i)}} a_i . \left[\mathrm{Pop}(i, l) \parallel S_{e(X\xi')}\right]_* \\
&\quad (+ \left[\mathbf{1} \parallel S_{e(X\xi')}\right]_*)_{h(X\xi')(\mathbf{1}) \geq 1}
\end{aligned}
$$

The stack may contain a series of transparent variables with multisets in which the occurrence of index i is strictly smaller than at the top. So, only the top element can be popped.

$$
\begin{aligned}
&= \sum_{i \in I_X} a_i . \tau . \left[\hat{X}(h(\xi')) \parallel S_{e(\xi')}\right]_* \\
&\quad + \sum_{i \in \mathcal{I}} \sum_{\substack{0 < l \leq h(X\xi')(i) \\ i \notin I_X \vee l \neq h(X\xi')(i)}} a_i . \left[\mathrm{Pop}(i, l) \parallel S_{e(X\xi')}\right]_* \\
&\quad (+ \left[\mathbf{1} \parallel S_{e(X\xi')}\right]_*)_{h(X\xi')(\mathbf{1}) \geq 1}
\end{aligned}
$$

Now, consider the second summation and optional summand. Given that $0 < l \leq h(X\xi')(i)$, it follows from the combination of Lemma 1 (in case $i \notin I_X$) or $l \neq h(X\xi)(i)$ (in case $i \in I_X$), that $0 < l \leq h(\xi')(i)$. Because we have forgetfulness of the stack $S_{e(X\xi')}$, it holds that $\left[\mathrm{Pop}(i, l) \parallel S_{e(X\xi')}\right]_* = \left[\mathrm{Pop}(i, l) \parallel S_{e(\xi')}\right]_*$ and that if $h(X\xi')(\mathbf{1}) \geq 1$, then $h(\xi')(\mathbf{1}) \geq 1$.

$$
\begin{aligned}
&= \sum_{i \in I_X} a_i . \tau . \left[\hat{X}(h(\xi')) \parallel S_{e(\xi')}\right]_* \\
&\quad + \sum_{i \in \mathcal{I}} \sum_{0 < l \leq h(\xi')(i)} a_i . \left[\mathrm{Pop}(i, l) \parallel S_{e(\xi')}\right]_* \\
&\quad (+ \left[\mathbf{1} \parallel S_{e(\xi')}\right]_*)_{h(\xi')(\mathbf{1}) \geq 1}
\end{aligned}
$$

Apply the definition of $\hat{X}(i, h(\xi'))$ on the first summation. Substitute the second summation and the optional summand with the definition of $\mathrm{Ctrl}(\xi')$.

$$= \sum_{i \in I_X} a_i.\tau.\left[\mathrm{Push}(\xi_i, h(\xi')) \parallel S_{e(\xi')}\right]_* + \left[\mathrm{Ctrl}(\xi') \parallel S_{e(\xi')}\right]_*$$

Perform $|\xi_i|$ pushes by repeatedly applying the definitions of $\mathrm{Push}(\xi, A)$ and $S_{e(\xi)}$.

$$\begin{aligned}
&= \sum_{i \in I_X} a_i.\tau^{|\xi_i|+1}.\left[\mathrm{Ctrl}(h(\xi_i\xi')) \parallel S_{e(\xi_i\xi')}\right]_* + \left[\mathrm{Ctrl}(\xi') \parallel S_{e(\xi')}\right]_* \\
&= \sum_{i \in I_X} a_i.\left[\mathrm{Ctrl}(h(\xi_i\xi')) \parallel S_{e(\xi_i\xi')}\right]_* + \left[\mathrm{Ctrl}(\xi') \parallel S_{e(\xi')}\right]_* \\
&= \sum_{i \in I_X} a_i.\overline{P}(\xi_i\xi') + \overline{P}(\xi').
\end{aligned}$$

This concludes our proof that there exists a, possibly infinite, recursive specification over MA that, in parallel with a partially forgetful stack, is equivalent to a context-free process P. □

In the following subsections, we will study under which conditions this specification reduces to a finite recursive specification over MA.

4.2 Opacity

In [18], context-free processes with $\mathbf{0}$ but without $\mathbf{1}$ were presented. Related to the absence of $\mathbf{1}$, we find that the intermediate solution reduces to a finite recursive specification, if none of the variables are transparent ($\mathcal{V}^{+\mathbf{1}} = \emptyset$), i.e. the specification is *opaque*.

From the specification of $\mathrm{Push}(\xi, A)$ we observe that now only sets are pushed on the stack (i.e. multisets in which each element occurs at most once). Hence, we can use a data set $D' = \mathcal{V} \times \mathcal{P}(\mathcal{I} \cup \{\mathbf{1}\})$ that no longer is infinite. We obtain a new, finite recursive specification, by replacing the equations for S, $\mathrm{Ctrl}(A)$, $\mathrm{Pop}(i, l)$ and $\mathrm{Push}(\xi, A)$ by the following ones:

$$\begin{aligned}
S &= \mathbf{1} + \sum_{\langle V, A\rangle \in D'} ?\langle V, A\rangle.S \cdot !\langle V, A\rangle.S, \\
\mathrm{Ctrl}(A) &= \sum_{i \in A} a_i.\mathrm{Pop}(i) \ (+\,\mathbf{1})_{\mathbf{1} \in A}, \\
\mathrm{Pop}(i) &= \sum_{\substack{\langle V, A\rangle \in D' \\ i \in I_V}} ?\langle V, A\rangle.\hat{V}(i, A), \\
\mathrm{Push}(\mathbf{1}, A) &= \mathrm{Ctrl}(A), \\
\mathrm{Push}(\xi Y, A) &= !\langle Y, A\rangle.\mathrm{Push}(\xi, I_Y).
\end{aligned}$$

Corollary 1. *For any context-free process P with recursive specification E that is opaque, there exists a regular process Q such that $P = \left[Q \parallel S_{e(X)}\right]_*$.*

4.3 Bounded Branching

Consider the following example, in which the variable Y is transparent.

$$X = a.X \cdot Y + b.\mathbf{1}, \qquad Y = \mathbf{1} + c.\mathbf{1}.$$

By executing a n times followed by b, the system gets to state Y^n. Here we have unbounded branching, since $Y^n \xrightarrow{c} Y^k$ for every $k < n$. This means state Y^n has n different outgoing c-steps, since none of the states Y^k are bisimilar. Thus, we cannot put a bound on the number of summands in the entire specification. The observation that the presence of $\mathbf{1}$-summands can cause unbounded branching is due to [8].

In case we have unbounded branching, it can be shown that there is no finite solution modulo rooted branching bisimulation. The reason for this, is that a regular process is certainly boundedly branching, so that the introduction of unbounded branching must take place through communication with the stack (in any solution, not only ours). This will result in internal τ transitions to states that are not rooted branching bisimilar, which makes that the τ transitions cannot be eliminated.

Assume now, that we have a specification for P that results in boundedly branching behavior, then the intermediate solution (see Sect. 4.1) does reduce to a finite recursive specification. In that case, the number of variables in a sequence ξ that can perform a certain action is bounded by some natural number N. The stack itself is an example of such a process. Hence, $h(\xi)(i) \leq N$ for any $i \in \mathcal{I}$, so the multisets in the data type D will never contain more than N occurrences for each index. We can reduce our specification by replacing $\mathrm{Ctrl}(A)$ by the following equation:

$$\mathrm{Ctrl}(A) = \sum_{i \in \mathcal{I}} \sum_{0 < l \leq A(i) \leq N} a_i.\mathrm{Pop}(i,l) \ (+\ \mathbf{1})_{A(\mathbf{1}) \geq 1}.$$

Corollary 2. *For any context-free process P with recursive specification E that has bounded branching, there exists a regular process Q such that $P = \left[Q \parallel S_{e(X)}\right]_*$.*

4.4 Unrestricted

In the previous subsection, we showed that there is no suitable pushdown automaton for the context-free process P, if P has unbounded branching. However, this observation relies on the fact that certain τ transitions cannot be eliminated. In this subsection, we show that the intermediate solution reduces to a finite recursive specification, for any P, if we accept the axiom of *contrasimulation* [12,19]:

$$a.(\tau.x + \tau.y) = a.x + a.y \qquad (a \in \mathcal{A}).$$

By this we weaken the equivalence on our transition systems. We do not know whether there is a stronger equivalence in the linear-time – branching-time spectrum II [12] for which a solution exists.

Starting from the intermediate solution (see Sect. 4.1), we can derive the following using the axiom of contrasimulation. In the first step, we use the

observation that, given some $i \in \mathcal{I}$ and $0 < l \leq h(\xi)(i)$, there exists a $\xi_{i,l}$ such that $\langle V(i), h(\xi_{i,l})\rangle e(\xi_{i,l})$ is a suffix of the stack contents $e(\xi)$, reachable through the forgetfulness of the stack. In the last step, we use the claim that $\sum_{0<l\leq h(\xi)(i)} \left[\mathrm{Pop}(i,l) \parallel S_{e(\xi)}\right]_* = \left[\mathrm{Pop}(i) \parallel S_{e(\xi)}\right]_*$.

$$\begin{aligned}
\left[\mathrm{Ctrl}(h(\xi)) \parallel S_{e(\xi)}\right]_* &= \sum_{i\in\mathcal{I}} \sum_{0<l\leq h(\xi)(i)} a_i.\left[\mathrm{Pop}(i,l) \parallel S_{e(\xi)}\right]_* \ (+\ \ldots) \\
&= \sum_{i\in\mathcal{I}} \sum_{0<l\leq h(\xi)(i)} a_i.\tau.\left[\hat{V(i)}(i,h(\xi_{i,l})) \parallel S_{e(\xi_{i,l})}\right]_* \ (+\ \ldots) \\
&= \sum_{\substack{i\in\mathcal{I}\\ h(\xi)(i)\geq 1}} a_i.\Big(\sum_{0<l\leq h(\xi)(i)} \tau.\tau.\left[\hat{V(i)}(i,h(\xi_{i,l})) \parallel S_{e(\xi_{i,l})}\right]_*\Big) \ (+\ \ldots) \\
&= \sum_{\substack{i\in\mathcal{I}\\ h(\xi)(i)\geq 1}} a_i.\Big(\sum_{0<l\leq h(\xi)(i)} \tau.\left[\hat{V(i)}(i,h(\xi_{i,l})) \parallel S_{e(\xi_{i,l})}\right]_*\Big) \ (+\ \ldots) \\
&= \sum_{\substack{i\in\mathcal{I}\\ h(\xi)(i)\geq 1}} a_i.\Big(\sum_{0<l\leq h(\xi)(i)} \left[\mathrm{Pop}(i,l) \parallel S_{e(\xi)}\right]_*\Big) \ (+\ \ldots) \\
&= \sum_{\substack{i\in\mathcal{I}\\ h(\xi)(i)\geq 1}} a_i.\left[\mathrm{Pop}(i) \parallel S_{e(\xi)}\right]_* \ (+\ \ldots).
\end{aligned}$$

We can reduce our specification by replacing $\mathrm{Ctrl}(A)$ and introducing $\mathrm{Pop}(i)$:

$$\begin{aligned}
\mathrm{Ctrl}(A) &= \sum_{\substack{i\in\mathcal{I}\\ A(i)\geq 1}} a_i.\mathrm{Pop}(i)\ (+\,\mathbf{1})_{A(\mathbf{1})\geq 1}, \\
\mathrm{Pop}(i) &= \sum_{\substack{\langle V,A\rangle\in D\\ i\in I_V}} ?\langle V,A\rangle.\hat{V}(i,A).
\end{aligned}$$

Finally, because we never inspect the multiplicity of an index in a multiset nor remove an element, we can replace multisets by sets and use $i \in A$ instead of $A(i) \geq 1$ and $\cup$ instead of $\uplus$.

Corollary 3. *For any context-free process P with recursive specification E, there exists a regular process Q such that $P = \left[Q \parallel S_{e(X)}\right]_*$, assuming the axiom of contrasimulation.*

5 Concluding Remarks

Every context-free process can be realized as a pushdown automaton. A pushdown automaton in concurrency theory is a regular process communicating with a stack.

We define a context-free process as the bisimulation equivalence class of a transition system given by a finite guarded recursive specification over Sequential Algebra. This algebra is needed for a full correspondence with automata theory, and includes constants $\mathbf{0}, \mathbf{1}$ not included in previous definitions of a context-free process.

The most difficult case is when the given context-free process has unbounded branching. This can only happen when a state of the system is given by a sequence of variables that have **1**-summands. In this case, there is no solution in rooted branching bisimulation semantics. We have found a solution in contrasimulation semantics, but do not know whether there are stronger equivalences in the spectrum of [12] for which a solution exists.

Concerning the reverse direction, not every regular process communicating with a stack is a context-free process. First of all, one must allow τ steps in the definition of context-free processes, because not all τ-steps of a pushdown automaton can be removed modulo rooted branching bisimulation or contrasimulation. Moreover, even if we allow τ steps, the theory of [17] shows that pushdown automata are more expressive than context-free processes without **1**. It is not trivial whether this result is still true when the expressivity of context-free processes is enlarged by adding termination. Research in this direction is left as future work.

The other famous result concerning context-free processes is the fact that bisimulation equivalence is decidable on this class, see [11]. Again, this result has been established for processes not including $\mathbf{0}, \mathbf{1}$. We expect that addition of **0** will not cause any difficulties, but addition of **1** will. We leave as an open problem whether bisimulation is decidable on the class of context-free processes as we have defined it.

Most questions concerning regular processes are settled, as we discussed in Sect. 2. A very important class of processes to be considered next are the computable processes. In [3], it was demonstrated that a Turing machine in concurrency theory can be presented as a regular process communicating with two stacks. By this means, it was established that every computable process can be realized as the abstraction of a solution of a finite guarded recursive specification over communication algebra. This result also holds in the presence of the constant **1**.

There are more classes of processes to be considered. The class of so-called *basic parallel processes* is given by finite guarded recursive specifications over Minimal Algebra extended with parallel composition (without communication). A prime example of such a process is the *bag*. Does the result of [10], that bisimulation is decidable on this class, still hold in the presence of **1**? Can we write every basic parallel process as a regular process communicating with a bag?

Acknowledgments

We would like to thank the members of the Formal Methods group, in particular Bas Luttik, for their comments, suggestions and vlaai.

The research of Van Tilburg was supported by the project "Models of Computation: Automata and Processes" (nr. 612.000.630) of the Netherlands Organization for Scientific Research (NWO).

References

1. Aceto, L., Fokkink, W.J., Verhoef, C.: Structural operational semantics. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 197–292. North-Holland, Amsterdam (2001)
2. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. Cambridge University Press, Cambridge (2008)
3. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: On the consistency of Koomen's fair abstraction rule. Theoretical Computer Science 51(1–2), 129–176 (1987)
4. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for processes generating context-free languages. Journal of the ACM 40(3), 653–682 (1993)
5. Baeten, J.C.M., Bravetti, M.: A ground-complete axiomatization of finite state processes in process algebra. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 246–262. Springer, Heidelberg (2005)
6. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press, Cambridge (1990)
7. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1/3), 109–137 (1984)
8. Bosscher, D.J.B.: Grammars modulo bisimulation. Ph.D. thesis. University of Amsterdam (1997)
9. Caucal, D.: Branching bisimulation for context-free processes. In: Shyamasundar, R.K. (ed.) FSTTCS 1992. LNCS, vol. 652, pp. 316–327. Springer, Heidelberg (1992)
10. Christensen, S., Hirshfeld, Y., Moller, F.: Bisimulation equivalence is decidable for basic parallel processes. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 143–157. Springer, Heidelberg (1993)
11. Christensen, S., Hüttel, H., Stirling, C.: Bisimulation equivalence is decidable for all context-free processes. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 138–147. Springer, Heidelberg (1992)
12. van Glabbeek, R.J.: The linear time – branching time spectrum ii. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
13. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3), 555–600 (1996)
14. Groote, J.F., Reniers, M.A.: Algebraic process verification. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 1151–1208. North-Holland, Amsterdam (2001)
15. Luttik, B.: Choice quantification in process algebra, Ph.D. thesis. University of Amsterdam (2002)
16. Moller, F.: The importance of the left merge operator in process algebras. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 752–764. Springer, Heidelberg (1990)
17. Moller, F.: Infinite results. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 195–216. Springer, Heidelberg (1996)
18. Srba, J.: Deadlocking states in context-free process algebra. In: Brim, L., Gruska, J., Zlatuska, J. (eds.) MFSC 1998. LNCS, vol. 1450, pp. 388–398. Springer, Heidelberg (1998)
19. Voorhoeve, M., Mauw, S.: Impossible futures and determinism. Information Processing Letters 80(1), 51–58 (2001)

Modeling Computational Security in Long-Lived Systems*

Ran Canetti[1,2], Ling Cheung[2], Dilsun Kaynar[3], Nancy Lynch[2], and Olivier Pereira[4]

[1] IBM T. J. Watson Research Center
[2] Massachusetts Institute of Technology
[3] Carnegie Mellon University
[4] Université catholique de Louvain

Abstract. For many cryptographic protocols, security relies on the assumption that adversarial entities have limited computational power. This type of security degrades progressively over the lifetime of a protocol. However, some cryptographic services, such as timestamping services or digital archives, are *long-lived* in nature; they are expected to be secure and operational for a very long time (i.e., super-polynomial). In such cases, security cannot be guaranteed in the traditional sense: a computationally secure protocol may become insecure if the attacker has a super-polynomial number of interactions with the protocol.

This paper proposes a new paradigm for the analysis of long-lived security protocols. We allow entities to be active for a potentially unbounded amount of real time, provided they perform only a polynomial amount of work *per unit of real time*. Moreover, the space used by these entities is allocated dynamically and must be polynomially bounded. We propose a new notion of *long-term implementation*, which is an adaptation of computational indistinguishability to the long-lived setting. We show that long-term implementation is preserved under polynomial parallel composition and exponential sequential composition. We illustrate the use of this new paradigm by analyzing some security properties of the long-lived timestamping protocol of Haber and Kamat.

1 Introduction

Computational security in long-lived systems: Security properties of cryptographic protocols typically hold only against resource-bounded adversaries. Consequently, mathematical models for representing and analyzing security of such protocols usually represent all participants as resource-bounded computational entities. The predominant way of formalizing such bounds is by representing

* Canetti's work on this project was supported by NSF award #CFF-0635297 and BSF award #2006317. Cheung and Lynch were supported by NSF Award #CCR-0326227. Kaynar was supported by US Army Research Office grant #DAAD19-01-1-0485. Pereira is a Research Associate of the F.R.S.-FNRS and was supported by the Belgian Interuniversity Attraction Pole P6/26 BCRYPT.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 114–130, 2008.

all entities as time-bounded machines, specifically, polynomial-time machines (a partial list of works representative of this direction includes [1,2,3,4,5]).

This modeling approach has been successful in capturing the security of protocols for many cryptographic tasks. However, it has a fundamental limitation: it assumes that the analyzed system runs for only a relatively "short" time. In particular, since all entities are polynomially-bounded (in the security parameter), the system's execution must end after a polynomial amount of time. This type of modeling is inadequate for analyzing security properties of protocols that are supposed to run for a "long" time, that is, an amount of time that is not bounded by a polynomial.

There are a number of natural tasks for which one would indeed be interested in the behavior of systems that run for a long time. Furthermore, a number of protocols have been developed for such tasks. Examples of such tasks include proactive security [6], forward secure signatures [7,8], forward secure encryption [7,9], and timestamping [10,11,12]. None of the existing models for analyzing security against computationally bounded adversaries is adequate for asserting and proving security properties of protocols for such "long-lived" tasks.

Related work: A first suggestion for an approach might be to use existing models, such as the PPT calculus [13], the Reactive Simulatability [14], or the Universally Composable security frameworks [3], with a sufficiently large value of the security parameter. However, this would be too limited for our purpose in that it would force protocols to protect against an overly powerful adversary *even in the short run,* while not providing any useful information in the long run. Similarly, turning to information theoretic security notions is not appropriate in our case because unbounded adversaries would be able to break computationally secure schemes instantaneously. We are interested in a notion of security that can protect protocols against an adversary that runs for a long time, but is only "reasonably powerful" at any point in time.

Recently, Müller-Quade and Unruh proposed a notion of *long-term security* for cryptographic protocols [15]. However, they consider adversaries that try to derive information from the protocol transcript *after* protocol conclusion. This work does not consider long-lived protocol execution and, in particular, the adversary of [15] has polynomially bounded interactions with the protocol parties, which is not suitable for the analysis of long-lived tasks such as those we described above.

Our approach: In this paper, we propose a new mathematical model for analyzing the security of such *long-lived systems.* To the best of our knowledge our work is the first one to tackle the issue of modeling computational security in long-lived systems. Our understanding of a long-lived system is that some protocol parties, including adversaries, may be active for an unbounded amount of real time, subject to the condition that only a polynomial amount of work can be done per unit of real time. Other parties may be active for only a short time, as in traditional settings. Thus, the adversary's interaction with the system is unbounded, and the adversary may perform an unbounded number of computation steps during the entire protocol execution. This renders traditional security

notions insufficient: computationally and even statistically secure protocols may fail if the adversary has unbounded interactions with the protocol.

Modeling long-lived systems requires significant departures from standard cryptographic modeling. First and foremost, unbounded entities cannot be modeled as *probabilistic polynomial time (PPT)* Turing machines. In search of a suitable alternative, we see the need to distinguish between two types of unbounded computation: steps performed steadily over a long period of time, versus those performed very rapidly in a short amount of time. The former conforms with our understanding of boundedness, while the latter does not. Guided by this intuition, we introduce real time explicitly into a basic probabilistic automata model, the Task-PIOA model [5], and impose computational restrictions in terms of *rates*, i.e., number of computation steps per unit of real time.

Another interesting challenge is the restriction on space, which traditionally is not an issue because PPT Turing machines can, by their nature, access only a polynomially bounded amount of space. In the long-lived setting, space restriction warrants explicit consideration. During the lifetime of a long-lived security protocol, we expect some components to die and other new ones to become active, for example, due to the use of cryptographic primitives that have a shorter life time than the protocol itself. Therefore, we find it important to be able to model dynamic allocation of space. We achieve this by restricting the use of state variables. In particular, all state variables of a dormant entity (either not yet invoked or already dead) are set to a special null value $\perp$. A system is regarded as bounded only if, at any point in its execution, only a bounded amount of space is needed to maintain all variables with non-$\perp$ values. For example, a sequential composition (in the temporal sense) of an unbounded number of entities is bounded if each entity uses a bounded amount of space.

Having appropriate restrictions on space and computation rates, we then define a new *long-term implementation relation*, $\leq_{\mathsf{neg,pt}}$, for long-lived systems. This is intended to extend the familiar notion of *computational indistinguishability*, where two systems (*real* and *ideal*) are deemed equivalent if their behaviors are indistinguishable from the point of view of a computationally bounded environment. However, notice that, in the long-lived setting, an environment with super-polynomial run time can typically distinguish the two systems trivially, e.g., by launching brute force attacks. This is true even if the environment has bounded computation rate. Therefore, our definition cannot rule out significant degradation of security in the overall lifetime of a system. Instead, we require that the *rate* of degradation is small at any point in time; in other words, the probability of a *new* successful attack during any polynomial-bounded window of time remains bounded during the lifetime of the system.

To capture this intuition, we extend the ideal systems traditionally used in cryptography by allowing them to take some designated *failure* steps, which allow an ideal system to take actions that could only occur in the real world, e.g., accepting forgeries as valid signatures, or producing ciphertexts that could allow recovering the corresponding plaintext. However, if failure steps do not occur

starting from some time t, then the ideal system starts following the specified ideal behavior.

Our long-term implementation relation $\leq_{\mathsf{neg,pt}}$ requires that the real system approximates the ideal's system's handling of failures. More precisely, we quantify over all real time points t and require that the real and ideal systems are computationally indistinguishable up to time $t+q$ (where q is polynomial in the security parameter), even if no failures steps are taken by the ideal system in the interval $[t, t+q]$. Notice that we do allow failure steps before time t. This expresses the idea that, despite any security breaches that may have occurred before time t, the success probability of a *fresh* attack in the interval $[t, t+q]$ is small. Our formal definition of $\leq_{\mathsf{neg,pt}}$ includes one more generalization: it considers failure steps in the real system as well as the ideal system, in both cases before the same real time t. This natural extension is intended to allow repeated use of $\leq_{\mathsf{neg,pt}}$, in verifying protocols using several levels of abstraction.

We show that $\leq_{\mathsf{neg,pt}}$ is transitive, and is preserved under the operations of polynomial parallel composition and exponential sequential composition. The sequential composition result highlights the power of our model to formulate and prove properties of an exponential number of entities in a meaningful way.

Example: Digital timestamping: As a proof of concept, we analyze some security properties of the digital timestamping protocol of Haber et al. [10,11,12], which was designed to address the problem of content integrity in long-term digital archives. In a nutshell, a digital timestamping scheme takes as input a document d at a specific time t_0, and produces a certificate c that can be used later to verify the existence of d at time t_0. The security requirement is that timestamp certificates are difficult to forge. Haber et al. note that it is inadvisable to use a single digital signature scheme to generate all timestamp certificates, even if signing keys are refreshed periodically. This is because, over time, any single signature scheme may be weakened due to advances in algorithmic research and/or discovery of vulnerabilities. Haber et al. propose a solution in which timestamps must be renewed periodically by generating a new certificate for the pair $\langle d, c\rangle$ using a new signature scheme. Thus, even if the signature scheme used to generate c is broken in the future, the new certificate c' still provides evidence that d existed at the time t_0 stated in the original certificate c.

We model the protocol of Haber et al. as the composition of a dispatcher component and a sequence of signature services. Each signature service "wakes up" at a certain time and is active for a specified amount of time before becoming dormant again. This can be viewed as a regular update of the signature service, which may entail a simple refresh of the signing key, or the adoption of a new signing algorithm. The dispatcher component accepts various timestamp requests and forwards them to the appropriate signature service. We show that the composition of the dispatcher and the signature services is indistinguishable from an ideal system, consisting of the same dispatcher composed with ideal signature functionalities. Specifically, this guarantees that the probability of a new forgery is small at any given point in time, regardless of any forgeries that may have happened in the past.

2 Task-PIOAs

We build our new framework using task-PIOAs [5], which are a version of Probabilistic Automata [16], augmented with an oblivious scheduling mechanism based on tasks. A task is a set of related actions (e.g., actions representing the same activity but with different parameters). We view tasks as basic groupings of events, both for real time scheduling and for imposing computational bounds (cf. Sections 3 and 4). In this section, we review basic notations related to task-PIOAs.

Notation: Given a set S, let $\mathsf{Disc}(S)$ denote the set of discrete probability measures on S. For $s \in S$, let $\delta(s)$ denote the *Dirac* measure on s, i.e., $\delta(s)(s) = 1$. Let V be a set of variables. Each $v \in V$ is associated with a *(static) type* $\mathsf{type}(v)$, which is the set of all possible values of v. We assume that $\mathsf{type}(v)$ is countable and contains the special symbol $\perp$. A *valuation* s for V is a function mapping every $v \in V$ to a value in $\mathsf{type}(v)$. The set of all valuations for V is denoted $val(V)$. Given $V' \subseteq V$, a valuation s' for V' is sometimes referred to as a *partial valuation* for V. Observe that s' induces a (full) valuation $\iota_V(s')$ for V, by assigning $\perp$ to every $v \notin V'$. Finally, for any set S with $\perp \notin S$, we write $S_\perp := S \cup \{\perp\}$.

PIOA: We define a *probabilistic input/output automaton (PIOA)* to be a tuple $\mathcal{A} = \langle V, S, s^{\mathsf{init}}, I, O, H, \Delta \rangle$, where:

(i) V is a set of *state variables* and $S \subseteq val(V)$ is a set of *states*;
(ii) $s^{\mathsf{init}} \in S$ is the *initial* state;
(iii) I, O and H are countable and pairwise disjoint sets of actions, referred to as *input, output and hidden actions*, respectively;
(iv) $\Delta \subseteq S \times (I \cup O \cup H) \times \mathsf{Disc}(S)$ is a *transition relation.*

The set $Act := I \cup O \cup H$ is the *action alphabet* of $\mathcal{A}$. If $I = \emptyset$, then $\mathcal{A}$ is said to be *closed*. The set of *external* actions of $\mathcal{A}$ is $I \cup O$ and the set of *locally controlled* actions is $O \cup H$. An *execution* is a sequence $\alpha = q_0 a_1 q_1 a_2 \ldots$ of alternating states and actions where $q_0 = s^{\mathsf{init}}$ and, for each $\langle q_i, a_{i+1}, q_{i+1} \rangle$, there is a transition $\langle q_i, a_{i+1}, \mu \rangle \in \Delta$ with $q_{i+1} \in \mathsf{Support}(\mu)$. A sequence obtained by restricting an execution of $\mathcal{A}$ to external actions is called a *trace*. We write $s.v$ for the value of variable v in state s. An action a is *enabled* in a state s if $\langle s, a, \mu \rangle \in \Delta$ for some μ. We require that $\mathcal{A}$ satisfy the following conditions.

- **Input Enabling:** For every $s \in S$ and $a \in I$, a is enabled in s.
- **Transition Determinism:** For every $s \in S$ and $a \in Act$, there is at most one $\mu \in \mathsf{Disc}(S)$ with $\langle s, a, \mu \rangle \in \Delta$. We write $\Delta(s, a)$ for such μ, if it exists.

Parallel composition for PIOAs is based on synchronization of shared actions. PIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be *compatible* if $V_i \cap V_j = Act_i \cap H_j = O_i \cap O_j = \emptyset$ whenever $i \neq j$. In that case, we define their *composition* $\mathcal{A}_1 \| \mathcal{A}_2$ to be $\langle V_1 \cup V_2, S_1 \times S_2, \langle s_1^{\mathsf{init}}, s_2^{\mathsf{init}} \rangle, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, H_1 \cup H_2, \Delta \rangle$, where Δ is the set of triples $\langle \langle s_1, s_2 \rangle, a, \mu_1 \times \mu_2 \rangle$ satisfying: (i) a is enabled in some s_i, and (ii) for every i, if $a \in Act_i$, then $\langle s_i, a, \mu_i \rangle \in \Delta_i$, otherwise $\mu_i = \delta(s_i)$. It is easy to check that input enabling and transition determinism are preserved under

composition. Moreover, the definition of composition can be generalized to any finite number of components.

Task-PIOA: To resolve nondeterminism, we make use of the notion of tasks introduced in [17,5]. Formally, a *task-PIOA* is a pair $\langle \mathcal{A}, \mathcal{R} \rangle$ where $\mathcal{A}$ is a PIOA and $\mathcal{R}$ is a partition of the locally-controlled actions of $\mathcal{A}$. The equivalence classes in $\mathcal{R}$ are called *tasks*. For notational simplicity, we often omit $\mathcal{R}$ and refer to the task-PIOA $\mathcal{A}$. The following additional axiom is assumed.

- **Action Determinism:** For every state s and every task T, at most one action $a \in T$ is enabled in s.

Unless otherwise stated, terminologies are inherited from the PIOA setting. For instance, if some $a \in T$ is enabled in a state s, then T is said to be *enabled* in s.

Example 1 (Clock automaton). Figure 1 describes a simple task-PIOA $\mathsf{Clock}(\mathbb{T})$, which has a $\mathsf{tick}(t)$ output action for every t in some discrete time domain $\mathbb{T}$. For concreteness, we assume that $\mathbb{T} = \mathbb{N}$, and write simply Clock. Clock has a single task tick, consisting of all $\mathsf{tick}(t)$ actions. These clock ticks are produced in order, for $t = 1, 2, \ldots$. In Section 3, we will define a mechanism that will ensure that each $\mathsf{tick}(t)$ occurs exactly at real time t.

$\mathsf{Clock}(\mathbb{T})$

Signature	**Tasks:** $\mathsf{tick} = \{\mathsf{tick}(*)\}$
Output: $\mathsf{tick}(t : \mathbb{T})$, $t > 0$	**States:** $count \in \mathbb{T}$, initially 0
Transitions	
$\mathsf{tick}(t)$	
Precondition: $count = t - 1$	Effect: $count := t$

Fig. 1. Task-PIOA Code for $\mathsf{Clock}(\mathbb{T})$

Operations: Given compatible task-PIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$, we define their *composition* to be $\langle \mathcal{A}_1 \| \mathcal{A}_2, \mathcal{R}_1 \cup \mathcal{R}_2 \rangle$. Note that $\mathcal{R}_1 \cup \mathcal{R}_2$ is an equivalence relation because compatibility requires disjoint sets of locally controlled actions. Moreover, it is easy to check that action determinism is preserved under composition.

We also define a *hiding* operator: given $\mathcal{A} = \langle V, S, s^{\mathsf{init}}, I, O, H, \Delta \rangle$ and $S \subseteq O$, $\mathsf{hide}(\mathcal{A}, S)$ is the task-PIOA given by $\langle V, S, s^{\mathsf{init}}, I, O', H', \Delta \rangle$, where $O' = O \setminus S$ and $H' = H \cup S$. This prevents other PIOAs from synchronizing with $\mathcal{A}$ via actions in S: any PIOA with an action in S in its signature is no longer compatible with $\mathcal{A}$.

Executions and traces: A *task schedule* for a closed task-PIOA $\langle \mathcal{A}, \mathcal{R} \rangle$ is a finite or infinite sequence $\rho = T_1, T_2, \ldots$ of tasks in $\mathcal{R}$. This induces a well-defined run of $\mathcal{A}$ as follows.

(i) From the start state s^{init}, we *apply* the first task T_1: due to action- and transition-determinism, T_1 specifies at most one transition from s^{init}; if such a transition exists, it is taken, otherwise nothing happens.
(ii) Repeat with remaining T_i's.

Such a run gives rise to a unique *probabilistic execution*, which is a probability distribution over executions in $\mathcal{A}$. For finite ρ, let $\mathsf{lstate}(\mathcal{A}, \rho)$ denote the state distribution of $\mathcal{A}$ after executing according to ρ. A state s is said to be *reachable* under ρ if $\mathsf{lstate}(\mathcal{A}, \rho)(s) > 0$. Moreover, the probabilistic execution induces a unique *trace distribution* $\mathsf{tdist}(\mathcal{A}, \rho)$, which is a probability distribution over the set of traces of $\mathcal{A}$. We refer the reader to [5] for more details on these constructions.

3 Real Time Scheduling Constraints

In this section, we describe how to model entities with unbounded lifetime but bounded processing rates. A natural approach is to introduce real time, so that computational restrictions can be stated in terms of the number of steps performed per unit real time. Thus, we define a *timed* task schedule τ for a closed task-PIOA $\langle \mathcal{A}, \mathcal{R} \rangle$ to be a finite or infinite sequence $\langle T_1, t_1 \rangle, \langle T_2, t_2 \rangle, \ldots$ such that: $T_i \in \mathcal{R}$ and $t_i \in \mathbb{R}_{\geq 0}$ for every i, and $t_1, t_2, \ldots$ is non-decreasing. Given a timed task schedule $\tau = \langle T_1, t_1 \rangle, \langle T_2, t_2 \rangle, \ldots$ and $t \in \mathbb{R}_{\geq 0}$, let $\mathsf{trunc}_{\geq t}(\tau)$ denote the result of removing all pairs $\langle T_i, t_i \rangle$ with $t_i \geq t$.

Following [18], we associate lower and upper real time bounds to each task. If l and u are, respectively, the lower bound and upper bound for a task T, then the amount of time between consecutive occurrences of T is at least l and at most u. To limit computational power, we impose a rate bound on the number of occurrences of T within an interval I, based on the length of I. A burst bound is also included for modeling flexibility.

Formally, a *bound map* for a task-PIOA $\langle \mathcal{A}, \mathcal{R} \rangle$ is a tuple $\langle \mathsf{rate}, \mathsf{burst}, \mathsf{lb}, \mathsf{ub} \rangle$ such that: (i) $\mathsf{rate}, \mathsf{burst}, \mathsf{lb} : \mathcal{R} \to \mathbb{R}_{\geq 0}$, (ii) $\mathsf{ub} : \mathcal{R} \to \mathbb{R}^{\infty}_{>0}$, and (iii) for all $T \in \mathcal{R}$, $\mathsf{lb}(T) \leq \mathsf{ub}(T)$. To ensure that rate and ub can be satisfied simultaneously, we require $\mathsf{rate}(T) \geq 1/\,\mathsf{ub}(T)$ whenever $\mathsf{rate}(T) \neq 0$ and $\mathsf{ub}(T) \neq \infty$. From this point on, we assume that every task-PIOA is associated with a particular bound map.

In the long version of this paper [19, Section 3], we formally define what it means for a timed task schedule τ to be valid for an interval under a given bound map. This definition states the technical conditions that simultaneously ensure that: (i) Consecutive appearances of a task T must be at least $\mathsf{lb}(T)$ apart, (ii) Consecutive appearances of a task T must be at most $\mathsf{ub}(T)$ apart, (iii) For any $d \in \mathbb{R}_{\geq 0}$ and any interval I' of length d, τ contains at most $\mathsf{rate}(T) \cdot d + \mathsf{burst}(T)$ elements with $\langle T, t \rangle$ with $t \in I'$.

Note that every timed schedule τ projects to an untimed schedule ρ by removing all real time information t_i, thereby inducing a trace distribution $\mathsf{tdist}(\mathcal{A}, \tau) := \mathsf{tdist}(\mathcal{A}, \rho)$.

In a parallel composition $\mathcal{A}_1 \| \mathcal{A}_2$, the composite bound map is the union of component bound maps: $\langle \mathsf{rate}_1 \cup \mathsf{rate}_2, \mathsf{burst}_1 \cup \mathsf{burst}_2, \mathsf{lb}_1 \cup \mathsf{lb}_2, \mathsf{ub}_1 \cup \mathsf{ub}_2 \rangle$.

Example 2 (Bound map for Clock*).* We use upper and lower bounds to ensure that Clock's internal counter evolves at the same rate as real time. Namely, we set $\mathsf{lb}(\mathsf{tick}) = \mathsf{ub}(\mathsf{tick}) = 1$. The rate and burst bounds are also set to 1. It is not hard to see that, regardless of the system of automata with which Clock is

composed, we always obtain the unique sequence $\langle \mathsf{tick}, 1\rangle, \langle \mathsf{tick}, 2\rangle, \ldots$ when we project a valid schedule to the task tick.

Note that we use real time solely to express constraints on task schedules. We do not allow computationally-bounded system components to maintain real-time information in their states, nor to communicate real-time information to each other. System components that require knowledge of time will maintain discrete approximations to time in their states, based on inputs from Clock.

4 Complexity Bounds

We are interested in modeling systems that run for an unbounded amount of real time. During this long life, we expect that a very large number of components will be active at various points in time, while only a small proportion of them will be active simultaneously. Defining complexity bounds in terms of the total number of components would then introduce unrealistic security constraints. Therefore, we find it more reasonable to define complexity bounds in terms of the characteristics of the components that are simultaneously active at any point in time.

To capture these intuitions, we define a notion of *step bound*, which limits the amount of computation a task-PIOA can perform, and the amount of space it can use, in executing a single step. By combining the step bound with the rate and burst bounds of Section 3, we obtain an *overall bound*, encompassing both bounded memory and bounded computation rates.

Note that we do not model situations where the rates of computation, or the computational power of machines, increases over time. This is an interesting direction in which the current research could be extended.

Step Bound: We assume some standard bit string encoding for Turing machines and for the names of variables, actions, and tasks. We also assume that variable valuations are encoded in the obvious way, as a list of name/value pairs. Let $\mathcal{A}$ be a task-PIOA with variable set V. Given state s, let $\hat{s}$ denote the partial valuation obtained from s by removing all pairs of the form $\langle v, \perp\rangle$. We have $\iota_V(\hat{s}) = s$, therefore no information is lost by reducing s to $\hat{s}$. This key observation allows us to represent a "large" valuation s with a "condensed" partial valuation $\hat{s}$.

Let $p \in \mathbb{N}$ be given. We say that a state s is p-bounded if the encoding of $\hat{s}$ is at most p bits long. The task-PIOA $\mathcal{A}$ is said to have *step bound* p if (a) the value of every variable is representable by at most p bits, (b) the name of every action name has length at most p bits, (c) the initial state s^{init} is p-bounded, (d) there are probabilistic Turing machines able to (i) determine which tasks are enabled in a given state of $\mathcal{A}$, (ii) determine which action a of a given task is enabled in a given state s of $\mathcal{A}$, and output a new state of $\mathcal{A}$ according to the distribution of $\Delta(s, a)$, (iii) determine if a candidate action a is an input action of $\mathcal{A}$ and, given a state s of $\mathcal{A}$, output a new state of $\mathcal{A}$ according to the distribution of $\Delta(s, a)$. Furthermore, those Turing Machines terminate after at most p steps on every input and they can be encoded using at most p bits.

Given a closed (i.e., no input actions) task-PIOA $\mathcal{A}$ with step bound p, one can easily define a Turing machine $M_{\mathcal{A}}$ with a combination of nondeterministic and probabilistic branching that simulates the execution of $\mathcal{A}$. It can be showed that the amount of work tape needed by $M_{\mathcal{A}}$ is polynomial in p.

It can also be shown that, when we compose task-PIOAs in parallel, the complexity of the composite is proportional to the sum of the component complexities. The proof is similar to that of the full version of [5, Lemma 4.2]. We also note that the hiding operator introduced in Section 2 preserves step bounds.

Overall Bound: We now combine real time bounds and step bounds. To do so, we represent global time using the clock automaton Clock (Figure 1). Let $p \in \mathbb{N}$ be given and let $\mathcal{A}$ be a task-PIOA compatible with Clock. We say that $\mathcal{A}$ is *p-bounded* if the following hold:

(i) $\mathcal{A}$ has step bound p.
(ii) For every task T of $\mathcal{A}$, $\mathsf{rate}(T)$ and $\mathsf{burst}(T)$ are both at most p.
(iii) For every $t \in \mathbb{N}$, let S_t denote the set of states s of $\mathcal{A} \| \mathsf{Clock}$ such that s is reachable under some valid schedule τ and $s.count = t$. There are at most p tasks T such that T is enabled in some $s \in S_t$. (Here, $s.count$ is the value of variable *count* of Clock in state s).

We say that $\mathcal{A}$ is *quasi-p-bounded* if $\mathcal{A}$ is of the form $\mathcal{A}' \| \mathsf{Clock}$ where $\mathcal{A}'$ is p-bounded.

Conditions (i) and (ii) are self-explanatory. Condition (iii) is a technical condition that ensures that the enabling of tasks does not change too rapidly. Without such a restriction, $\mathcal{A}$ could cycle through a large number of tasks between two clock ticks, without violating the rate bound of any individual task.

Task-PIOA Families: We now extend our definitions to task-PIOA families, indexed by a *security parameter* k. More precisely, a *task-PIOA family* $\bar{\mathcal{A}}$ is an indexed set $\{\mathcal{A}_k\}_{k \in \mathbb{N}}$ of task-PIOAs. Given $p : \mathbb{N} \to \mathbb{N}$, we say that $\bar{\mathcal{A}}$ is *p-bounded* just in case: for all k, $\mathcal{A}_k$ is $p(k)$-bounded. If p is a polynomial, then we say that $\bar{\mathcal{A}}$ is *polynomially bounded*. The notions of compatibility and parallel composition for task-PIOA families are defined pointwise. We now present an example of a polynomially bounded family of task-PIOAs—a signature service that we use in our digital timestamping example. The complete formal specification for these task-PIOAs can be found in the long version of this paper [19].

Example 3 (Signature Service). A *signature scheme* Sig consists of three algorithms: KeyGen, Sign and Verify. KeyGen is a probabilistic algorithm that outputs a signing-verification key pair $\langle sk, vk \rangle$. Sign is a probabilistic algorithm that produces a signature σ from a message m and the key sk. Finally, Verify is a deterministic algorithm that maps $\langle m, \sigma, vk \rangle$ to a boolean. The signature σ is said to be *valid* for m and vk if $\mathsf{Verify}(m, \sigma, vk) = 1$.

Let *SID* be a domain of service identifiers. For each $j \in SID$, we build a signature service as a family of task-PIOAs indexed by security parameter k. Specifically, we define three task-PIOAs, $\mathsf{KeyGen}(k, j)$, $\mathsf{Signer}(k, j)$, and $\mathsf{Verifier}(k, j)$ for every pair $\langle k, j \rangle$, representing the key generator, signer, and verifier, respectively. The composition of these three task-PIOAs gives a signature service. We

assume a function $\mathsf{alive} : \mathbb{T} \to 2^{SID}$ such that, for every t, $\mathsf{alive}(t)$ is the set of services alive at discrete time t. The lifetime of each service j is then given by $\mathsf{aliveTimes}(j) := \{t \in \mathbb{T} | j \in \mathsf{alive}(t)\}$; we assume this to be a finite set of consecutive numbers.

Assuming the algorithms KeyGen_j, Sign_j and Verify_j are polynomial time, it not hard to check that the composite $\mathsf{KeyGen}(k,j)\|\mathsf{Signer}(k,j)\|\mathsf{Verifier}(k,j)$ has step bound $p(k)$ for some polynomial p. If $\mathsf{rate}(T)$ and $\mathsf{burst}(T)$ are at most $p(k)$ for every T, then the composite is $p(k)$-bounded. The family $\{\mathsf{KeyGen}(k,j)\| \mathsf{Signer}(k,j)\|\mathsf{Verifier}(k,j)\}_{k\in\mathbb{N}}$ is therefore polynomially bounded.

5 Long-Term Implementation Relation

Much of modern cryptography is based on the notion of computational indistinguishability. For instance, an encryption algorithm is (chosen-plaintext) secure if the ciphertexts of two distinct but equal-length messages are indistinguishable from each other, even if the plaintexts are generated by the distinguisher itself. The key assumption is that the distinguisher is computationally bounded, so that it cannot launch a brute force attack. In this section, we adapt this notion of indistinguishability to the long-lived setting.

We define an implementation relation based on closing environments and acceptance probabilities. Let $\mathcal{A}$ be a closed task-PIOA with output action acc and task $\mathsf{acc} = \{\mathsf{acc}\}$. Let τ be a timed task schedule for $\mathcal{A}$. The *acceptance probability* of $\mathcal{A}$ under τ is: $\mathbf{P}_{\mathsf{acc}}(\mathcal{A},\tau) := \Pr[\beta \text{ contains } \mathsf{acc} : \beta \leftarrow_{\mathsf{R}} \mathsf{tdist}(\mathcal{A},\tau)]$; that is, the probability that a trace drawn from the distribution $\mathsf{tdist}(\mathcal{A},\tau)$ contains the action acc. If $\mathcal{A}$ is not necessarily closed, we include a closing environment. A task-PIOA Env is an *environment* for $\mathcal{A}$ if it is compatible with $\mathcal{A}$ and $\mathcal{A}\|\mathsf{Env}$ is closed. From here on, we assume that every environment has output action acc.

In the short-lived setting, we say that a system $\mathcal{A}_1$ implements another system $\mathcal{A}_2$ if every run of $\mathcal{A}_1$ can be "matched" by a run of $\mathcal{A}_2$ such that no polynomial time environment can distinguish the two runs. As we discussed in the introduction, this type of definition is too strong for the long-lived setting, because we must allow environments with unbounded total run time (as long as they have bounded rate and space).

For example, consider the timestamping protocol of [11,12] described in Section 1. After running for a long period of real time, a distinguisher environment may be able to forge a signature with non-negligible probability. As a result, it can distinguish the real system from an ideal timestamping system, in the traditional sense. However, the essence of the protocol is that such failures can in fact be tolerated, because they do not help the environment to forge *new* signatures, after a new, uncompromised signature service becomes active.

This timestamping example suggests that we need a new notion of long-term implementation that makes meaningful security guarantees in any polynomial-bounded window of time, in spite of past security failures. Our new implementation relation aims to capture this intuition.

First we define a comparability condition for task-PIOAs: $\mathcal{A}^1$ and $\mathcal{A}^2$ are said to be *comparable* if they have the same external interface, that is, $I^1 = I^2$ and $O^1 = O^2$. In this case, every environment E for $\mathcal{A}^1$ is also an environment for $\mathcal{A}^2$, provided E is compatible with $\mathcal{A}^2$.

Let $\mathcal{A}^1$ and $\mathcal{A}^2$ be comparable task-PIOAs. To model security failure events in both automata, we let F^1 be a set of designated *failure tasks* of $\mathcal{A}^1$, and let F^2 be a set of *failure tasks* of A^2. We assume that each task in F^1 and F^2 has ∞ as its upper bound.

Given $t \in \mathbb{R}_{\geq 0}$ and an environment Env for both $\mathcal{A}^1$ and $\mathcal{A}^2$, we consider two experiments. In the first experiment, Env interacts with $\mathcal{A}^1$ according to some valid task schedule τ_1 of $\mathcal{A}^1 \| \mathsf{Env}$, where τ_1 does not contain any tasks from F^1 from time t onwards. In the second experiment, Env interacts with $\mathcal{A}^2$ according to some valid task schedule τ_2 of $\mathcal{A}^2 \| \mathsf{Env}$, where τ_2 does not contain any tasks from F^2 from time t onwards. Our definition requires that the first experiment "approximates" the second one, that is, if $\mathcal{A}^1$ acts ideally (does not perform any of the failure tasks in F^1) after time t, then it simulates $\mathcal{A}^2$, also acting ideally from time t onwards.

More specifically, we require that, for any valid τ_1, there exists a valid τ_2 as above such that the two executions are identical before time t from the point of view of the environment. That is, the probabilistic execution is the same before time t. Moreover, the two executions are overall *computationally indistinguishable*, namely, the difference in acceptance probabilities in these two experiments is negligible provided Env is computationally bounded.

If τ is a schedule of $\mathcal{A} \| \mathcal{B}$, then we define $\mathsf{proj}_{\mathcal{B}}(\tau)$ to be the result of removing all $\langle T_i, t_i \rangle$ where T_i is *not* a task of $\mathcal{B}$. Moreover, let $\mathsf{Execs}_{\mathcal{B}}(\mathcal{A} \| \mathcal{B}, \tau)$ denote the distribution of executions of $\mathcal{B}$ when executed with $\mathcal{A}$ under schedule τ.

Definition 1. *Let $\mathcal{A}^1$ and $\mathcal{A}^2$ be comparable task-PIOAs that are both compatible with* Clock*. Let F^1 and F^2 be sets of tasks of, respectively, $\mathcal{A}^1$ and $\mathcal{A}^2$, such that for any $T \in (F^1 \cup F^2)$, $\mathsf{ub}(T) = \infty$. Let $p, q \in \mathbb{N}$ and $\epsilon \in \mathbb{R}_{\geq 0}$ be given. Then we say that $(\mathcal{A}^1, F^1) \leq_{p,q,\epsilon} (\mathcal{A}^2, F^2)$ provided that the following is true: For every $t \in \mathbb{R}_{\geq 0}$, every quasi-p-bounded environment* Env*, and every valid timed schedule τ_1 for $\mathcal{A}^1 \| \mathsf{Env}$ for the interval $[0, t+q]$ that does not contain any pairs of the form $\langle T_i, t_i \rangle$ where $T_i \in F^1$ and $t_i \geq t$, there exists a valid timed schedule τ_2 for $\mathcal{A}^2 \| \mathsf{Env}$ for the interval $[0, t+q]$ such that:*

(i) $\mathsf{proj}_{\mathsf{Env}}(\tau_1) = \mathsf{proj}_{\mathsf{Env}}(\tau_2)$*;*
(ii) *τ_2 does not contain any pairs of the form $\langle T_i, t_i \rangle$ where $T_i \in F^2$ and $t_i \geq t$;*
(iii) $\mathsf{Execs}_{\mathsf{Env}}(\mathcal{A}^1 \| \mathsf{Env}, \mathsf{trunc}_{\geq t}(\tau_1)) = \mathsf{Execs}_{\mathsf{Env}}(\mathcal{A}^2 \| \mathsf{Env}, \mathsf{trunc}_{\geq t}(\tau_2))$*;*
(iv) $|\mathbf{P}_{\mathsf{acc}}(\mathcal{A}^1 \| \mathsf{Env}, \tau_1) - \mathbf{P}_{\mathsf{acc}}(\mathcal{A}^2 \| \mathsf{Env}, \tau_2)| \leq \epsilon$*.*

It can be observed that the $\leq_{p,q,\epsilon}$ is transitive up to additive errors [19].

The relation $\leq_{p,q,\epsilon}$ can be extended to task-PIOA families as follows. Let $\bar{\mathcal{A}}^1 = \{(\bar{\mathcal{A}}^1)_k\}_{k \in \mathbb{N}}$ and $\bar{\mathcal{A}}^2 = \{(\bar{\mathcal{A}}^2)_k\}_{k \in \mathbb{N}}$ be pointwise comparable task-PIOA families. Let $\bar{F}^1$ be a family of sets such that each $(\bar{F}^1)_k$ is a set of tasks of $(\bar{\mathcal{A}}^1)_k$ and let $\bar{F}^2$ be a family of sets such that each $(\bar{F}^2)_k$ is a set of tasks of $(\bar{\mathcal{A}}^2)_k$,

satisfying the condition that each task of those sets has an infinite upper bound. Let $\epsilon : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $p, q : \mathbb{N} \to \mathbb{N}$ be given. We say that $(\bar{\mathcal{A}}^1, \bar{F}^1) \leq_{p,q,\epsilon} (\bar{\mathcal{A}}^2, \bar{F}^2)$ just in case $((\bar{\mathcal{A}}^1)_k, (\bar{F}^1)_k) \leq_{p(k),q(k),\epsilon(k)} ((\bar{\mathcal{A}}^2)_k, (\bar{F}^2)_k)$ for every k.

Restricting our attention to negligible error and polynomial time bounds, we obtain the long-term implementation relation $\leq_{\mathsf{neg,pt}}$. Formally, a function $\epsilon : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is said to be *negligible* if, for every constant $c \in \mathbb{N}$, there exists $k_0 \in \mathbb{N}$ such that $\epsilon(k) < \frac{1}{k^c}$ for all $k \geq k_0$. (That is, ϵ diminishes more quickly than the reciprocal of any polynomial.) Given task-PIOA families $\bar{\mathcal{A}}^1$ and $\bar{\mathcal{A}}^2$ and task set families $\bar{F}^1$ and $\bar{F}^2$, respectively, of $\bar{\mathcal{A}}^1$ and $\bar{\mathcal{A}}^2$, we say that $(\bar{\mathcal{A}}^1, \bar{F}^1) \leq_{\mathsf{neg,pt}} (\bar{\mathcal{A}}^2, \bar{F}^2)$ if $\forall p, q\ \exists \epsilon : (\bar{\mathcal{A}}^1, \bar{F}^1) \leq_{p,q,\epsilon} (\bar{\mathcal{A}}^2, \bar{F}^2)$, where p, q are polynomials and ϵ is a negligible function.

Example 4 (Ideal Signature Functionality). In order to illustrate the use of the relation $\leq_{\mathsf{neg,pt}}$ in our example, we specify an *ideal signature functionality* $\mathsf{SigFunc}$, and show that it is implemented by the real signature service of Section 4.

As with KeyGen, Signer, and $\mathsf{Verifier}$, each instance of $\mathsf{SigFunc}$ is parameterized with a security parameter k and an identifier j. It is very similar to the composition of $\mathsf{Signer}(k, j)$ and $\mathsf{Verifier}(k, j)$. The important difference is that $\mathsf{SigFunc}(k, j)$ maintains an additional variable *history*, which records the set of signed messages. In addition, $\mathsf{SigFunc}(k, j)$ has an internal action fail_j, which sets a boolean flag *failed*. If *failed* = false, then $\mathsf{SigFunc}(k, j)$ uses *history* to answer verification requests: a signature is rejected if the submitted message is not in *history*, even if Verify_j returns 1. If *failed* = true, then $\mathsf{SigFunc}(k, j)$ bypasses the check on *history*, so that its answers are identical to those from the real signature service.

Let us define $\mathsf{RealSig}(j)_k = \mathsf{hide}(\mathsf{KeyGen}(k, j) \| \mathsf{Signer}(k, j) \| \mathsf{Verifier}(k, j), \mathsf{signKey}_j)$ and $\mathsf{IdealSig}(j)_k = \mathsf{hide}(\mathsf{KeyGen}(k, j) \| \mathsf{SigFunc}(k, j), \mathsf{signKey}_j)$. We define families from those automata in the obvious way: $\overline{\mathsf{RealSig}} := \{\mathsf{RealSig}_k\}_{k \in \mathbb{N}}$ and $\overline{\mathsf{IdealSig}} := \{\mathsf{IdealSig}_k\}_{k \in \mathbb{N}}$. We show that the real signature service implements the ideal signature functionality. The proof, which relies on a reduction to standard properties of a signature scheme, can be found in [19].

Theorem 1. *Let $j \in SID$ be given. Suppose that $\langle \mathsf{KeyGen}_j, \mathsf{Sign}_j, \mathsf{Verify}_j \rangle$ is a complete and EUF-CMA secure signature scheme. Then $(\overline{\mathsf{RealSig}}(j), \{\}) \leq_{\mathsf{neg,pt}} (\overline{\mathsf{IdealSig}}(j), \{\mathsf{fail}_j\})$.*

6 Composition Theorems

In practice, cryptographic services are seldom used in isolation. Usually, different types of services operate in conjunction, interacting with each other and with multiple protocol participants. For example, a participant may submit a document to an encryption service to obtain a ciphertext, which is later submitted to a timestamping service. In such situations, it is important that the services are provably secure even in the context of composition.

In this section, we consider two types of composition. The first, *parallel composition*, is a combination of services that are active at the same time and may

interact with each other. Given a polynomially bounded collection of real services such that each real service implement some ideal service, the parallel composition of the real services is guaranteed to implement that of the ideal services.

The second type, *sequential composition*, is a combination of services that are active in succession. The interaction between two distinct services is much more limited in this setting, because the earlier one must have finished execution before the later one begins. An example of such a collection is the signature services in the timestamping protocol of [12,11], where each service is replaced by the next at regular intervals.

As in the parallel case, we prove that the sequential composition of real services implements the sequential composition of ideal services. We are able to relax the restriction on the number of components from polynomial to exponential.[1] This highlights a unique aspect of our implementation relation: essentially, from any point t on the real time line, we focus on a polynomial length interval starting from t.

Parallel Composition: Using a standard hybrid argument, as exemplified in [20] for instance, it is possible to show that the relation $\leq_{\mathsf{neg,pt}}$ is preserved under polynomial parallel composition. The theorem contains a technicality: instead of simply assuming $\leq_{\mathsf{neg,pt}}$ relationships for all the components, we assume a slightly stronger property, in which the same negligible function ϵ is assumed for all of the components; that is, ϵ is not allowed to depend on the component index i.

Theorem 2 (Parallel Composition Theorem for $\leq_{\mathsf{neg,pt}}$). *Let $\bar{\mathcal{A}}_1^1, \bar{\mathcal{A}}_2^1, \ldots$ and $\bar{\mathcal{A}}_1^2, \bar{\mathcal{A}}_2^2, \ldots$ be two infinite sequences of task-PIOA families, with $\bar{\mathcal{A}}_i^1$ comparable to $\bar{\mathcal{A}}_i^2$ for every i. Suppose that $\bar{\mathcal{A}}_1^{\alpha_1}, \bar{\mathcal{A}}_2^{\alpha_2}, \ldots$ are pairwise compatible for any combination of $\alpha_i \in \{1,2\}$. Let b be any polynomial, and for each k, let $(\widehat{\mathcal{A}}^1)_k$ and $(\widehat{\mathcal{A}}^2)_k$ denote $\|_{i=1}^{b(k)}(\bar{\mathcal{A}}_i^1)_k$ and $\|_{i=1}^{b(k)}(\bar{\mathcal{A}}_i^2)_k$, respectively. Let r and s be polynomials, $r, s : \mathbb{N} \to \mathbb{N}$, such that r is nondecreasing, and for every i, k, both $(\bar{\mathcal{A}}_i^1)_k$ and $(\bar{\mathcal{A}}_i^2)_k$ are bounded by $s(k) \cdot r(i)$.*

For each i, let $\bar{F}_i^1$ be a family of sets such that $(\bar{F}_i^1)_k$ is a set of tasks of $(\bar{\mathcal{A}}_i^1)_k$ for every k, and let $\bar{F}_i^2$ be a family of sets such that $(\bar{F}_i^2)_k$ is a set of tasks of $(\bar{\mathcal{A}}_i^2)_k$ for every k, where all these tasks have infinite upper bounds. Let $(\hat{F}^1)_k$ and $(\hat{F}^2)_k$ denote $\bigcup_{i=1}^{b(k)} (\bar{F}_i^1)_k$ and $\bigcup_{i=1}^{b(k)} (\bar{F}_i^2)_k$, respectively.

Assume:

$$\forall p, q \ \exists \epsilon \ \forall i \ (\bar{\mathcal{A}}_i^1, \bar{F}_i^1) \leq_{p,q,\epsilon} (\bar{\mathcal{A}}_i^2, \bar{F}_i^2), \tag{1}$$

where p, q are polynomials and ϵ is a negligible function.

Then $(\widehat{\mathcal{A}}^1, \hat{F}^1) \leq_{\mathsf{neg,pt}} (\widehat{\mathcal{A}}^2, \hat{F}^2)$.

Sequential Composition: We now treat the more interesting case, namely, exponential sequential composition. The first challenge is to formalize the notion of sequentiality. On a syntactic level, all components in the collection are combined using

[1] In our model, it is not meaningful to exceed an exponential number of components, because the length of the description of each component is polynomially bounded.

the parallel composition operator. To capture the idea of successive invocation, we introduce some auxiliary notions. Intuitively, we distinguish between *active* and *dormant* entities. Active entities may perform actions and store information in memory. Dormant entities have no available memory and do not enable locally controlled actions.[2] In Definition 2, we formalize the idea of an entity $\mathcal{A}$ being active during a particular time interval. Then we introduce sequentiality in Definition 3.

Definition 2. *Let $\mathcal{A}$ be a task-PIOA and let reals $t_1 \leq t_2$ be given. We say that $\mathcal{A}$ is* restricted to *the interval $[t_1, t_2]$ if for every $t \notin [t_1, t_2]$, environment* Env *for $\mathcal{A}$ of the form* $\mathsf{Env}' \| \mathsf{Clock}$*, valid schedule τ for $\mathcal{A} \| \mathsf{Env}$ for $[0, t]$, and state s reachable under τ, no locally controlled actions of $\mathcal{A}$ are enabled in s, and $s.v = \perp$ for every variable v of $\mathcal{A}$.*

Definition 3 (Sequentiality). *Let $\mathcal{A}_1, \mathcal{A}_2, \ldots$ be pairwise compatible task-PIOAs. We say that $\mathcal{A}_1, \mathcal{A}_2, \ldots$ are* sequential with respect to *the the nondecreasing sequence $t_1, t_2, \ldots$ of nonnegative reals provided that for every i, $\mathcal{A}_i$ is restricted to $[t_i, t_{i+1}]$.*

Note the slight technicality that each $\mathcal{A}_i$ may overlap with $\mathcal{A}_{i+1}$ at the boundary time t_{i+1}.

Theorem 3 (Sequential Composition Theorem for $\leq_{\mathsf{neg,pt}}$). *Let $\bar{\mathcal{A}}_1^1, \bar{\mathcal{A}}_2^1, \ldots$ and $\bar{\mathcal{A}}_1^2, \bar{\mathcal{A}}_2^2, \ldots$ be two infinite sequences of task-PIOA families, with $\bar{\mathcal{A}}_i^1$ comparable to $\bar{\mathcal{A}}_i^2$ for every i. Suppose that $\bar{\mathcal{A}}_1^{\alpha_1}, \bar{\mathcal{A}}_2^{\alpha_2}, \ldots$ are pairwise compatible for any combination of $\alpha_i \in \{1, 2\}$. Let $L : \mathbb{N} \to \mathbb{N}$ be an exponential function and, for each k, let $(\widehat{\mathcal{A}}^1)_k$ and $(\widehat{\mathcal{A}}^2)_k$ denote $\|_{i=1}^{L(k)} (\bar{\mathcal{A}}_i^1)_k$ and $\|_{i=1}^{L(k)} (\bar{\mathcal{A}}_i^2)_k$, respectively. Let $\hat{p}$ be a polynomial such that both $\widehat{\mathcal{A}}^1$ and $\widehat{\mathcal{A}}^2$ are $\hat{p}$-bounded.*

Suppose there exists an increasing sequence of nonnegative reals $t_1, t_2, \ldots$ such that, for each k, both $(\bar{\mathcal{A}}_1^1)_k, \ldots, (\bar{\mathcal{A}}_{L(k)}^1)_k$ and $(\bar{\mathcal{A}}_1^2)_k, \ldots, (\bar{\mathcal{A}}_{L(k)}^2)_k$ are sequential for $t_1, t_2, \ldots$ Assume there is a constant real number c such that consecutive t_i's are at least c apart.

For each i, let $\bar{F}_i^1$ be a family of sets such that $(\bar{F}_i^1)_k$ is a set of tasks of $(\bar{\mathcal{A}}_i^1)_k$ for every k and let $\bar{F}_i^2$ be a family of sets such that $(\bar{F}_i^2)_k$ is a set of tasks of $(\bar{\mathcal{A}}_i^2)_k$ for every k, where all these tasks have infinite upper bounds. Let $(\hat{F}^1)_k$ and $(\hat{F}^2)_k$ denote $\bigcup_{i=1}^{L(k)} (\bar{F}_i^1)_k$ and $\bigcup_{i=1}^{L(k)} (\bar{F}_i^2)_k$, respectively.

Assume:

$$\forall p, q \; \exists \epsilon \; \forall i \; (\bar{\mathcal{A}}_i^1, \bar{F}_i^1) \leq_{p,q,\epsilon} (\bar{\mathcal{A}}_i^2, \bar{F}_i^2), \tag{2}$$

where p, q are polynomials and ϵ is a negligible function.

Then $(\widehat{\mathcal{A}}^1, \hat{F}^1) \leq_{\mathsf{neg,pt}} (\widehat{\mathcal{A}}^2, \hat{F}^2)$.

This sequential composition theorem can be easily extended to the case where a bounded number of components of the system are active concurrently [19].

Application to Digital Timestamping: In this section, we present a formal model of the digital timestamping protocol of Haber et al. (cf. Section 1). Recall

[2] For technical reasons, dormant entities must synchronize on input actions. Some inputs cause dormant entities to become active, while all others are trivial loops on the null state.

the real and ideal signature services from Sections 4 and 5. The timestamping protocol consists of a dispatcher component and a collection of real signature services. Similarly, the ideal protocol consists of the same dispatcher with a collection of ideal signature services. Using the sequential composition theorem (Thm. 3) and its extension to a bounded number of concurrent components, we prove that the real protocol implements the ideal protocol with respect to the long-term implementation relation $\leq_{\mathsf{neg,pt}}$. This result implies that, no matter what security failures (forgeries, guessed keys, etc.) occur up to any particular time t, new certifications and verifications performed by services that awaken after time t will still be correct (with high probability) for a polynomial-length interval of time after t.

Note that this result does *not* imply that any particular document is reliably certified for super-polynomial time. In fact, Haber's protocol does not guarantee this: even if a document certificate is refreshed frequently by new services, there is at any time a small probability that the environment guesses the current certificate, thus creating a forgery. That probability, over super-polynomial time, becomes large. Once the environment guesses a current certificate, it can continue to refresh the certificate forever, thus maintaining the forgery.

Dispatcher: We define $\mathsf{Dispatcher}_k$ for each security parameter k and set *SID*, the domain of service names, to be $\mathbb{N}$. If the environment sends a first-time certificate request, $\mathsf{Dispatcher}_k$ requests a signature from signature service j, where j is the service active at the time where this request is transmitted. After service j returns the new certificate, $\mathsf{Dispatcher}_k$ transmits it to the environment.

If a renew request for a certificate issued by the j-th signing service comes in, $\mathsf{Dispatcher}_k$ first checks to see if service j is still usable. If not, it sends a notification to the environment. Otherwise, it asks the j-th signature verification service to check the validity of the certificate. If service j answers affirmatively, $\mathsf{Dispatcher}_k$ sends a signature request to the j'-th signature service, active at the time of this request. When service j' returns, $\mathsf{Dispatcher}_k$ issues a new certificate to the environment.

Assume the following concrete time scheme. Let d be a positive natural number. Each service j is in $\mathsf{alive}(t)$ for $t = (j-1)d, \ldots, (j+2)d-1$, so j is alive in the real time interval $[(j-1)d, (j+2)d]$. Thus, at any real time t, at most three services are concurrently alive; more precisely, t lies in the interior of the intervals for at most three services. Besides, signature service j accepts signature requests for $t = (j-1)d, \ldots, jd-1$.

Protocol Correctness: For every security parameter k, let $SID_k \subseteq SID$ denote the set of $p(k)$-bit numbers, for some polynomial p. Recall from Section 5 that $\mathsf{RealSig}(j)_k = \mathsf{hide}(\mathsf{KeyGen}(k,j) \| \mathsf{Signer}(k,j) \| \mathsf{Verifier}(k,j), \mathsf{signKey}_j)$ and $\mathsf{IdealSig}(j)_k = \mathsf{hide}(\mathsf{KeyGen}(k,j) \| \mathsf{SigFunc}(k,j), \mathsf{signKey}_j)$. Here we define $\mathsf{Real}_k = \|_{j \in SID_k} \mathsf{RealSig}(j)_k$, $\mathsf{Ideal}_k = \|_{j \in SID_k} \mathsf{IdealSig}(j)_k$, and $\mathsf{RealSigSys}_k := \mathsf{Dispatcher}_k \| \mathsf{Real}_k$, $\mathsf{IdealSigSys}_k := \mathsf{Dispatcher}_k \| \mathsf{Ideal}_k$. Eventually, define $\overline{\mathsf{Real}} := \{\mathsf{Real}_k\}_{k \in \mathbb{N}}$, $\overline{\mathsf{Ideal}} := \{\mathsf{Ideal}_k\}_{k \in \mathbb{N}}$, $\overline{\mathsf{RealSigSys}} := \{\mathsf{RealSigSys}_k\}_{k \in \mathbb{N}}$ and $\overline{\mathsf{IdealSigSys}} := \{\mathsf{IdealSigSys}_k\}_{k \in \mathbb{N}}$. We show the following theorem.

Theorem 4. *Assume the concrete time scheme described above and assume that every signature scheme used in the timestamping protocol is complete and existentially unforgeable. Then* $(\overline{\mathsf{RealSigSys}}, \emptyset) \leq_{\mathsf{neg,pt}} (\overline{\mathsf{IdealSigSys}}, \bar{F})$*, where* $\bar{F}_k := \bigcup_{j \in SID_k} \{\{\mathsf{fail}_j\}\}$ *for every* k*.*

In order to prove this theorem, we first observe that certain components of the real and ideal systems are restricted to certain time intervals, in the sense of Definition 2: at most three $\mathsf{RealSig}(i)_k$ and $\mathsf{IdealSig}(i)_k$ services are alive at the same time. Then, we observe that the task-PIOA families $\overline{\mathsf{Real}}$ and $\overline{\mathsf{Ideal}}$ are polynomially bounded and apply the extension of our sequential composition theorem (Thm. 3) for bounded concurrency to show that $(\overline{\mathsf{Real}}, \emptyset) \leq_{\mathsf{neg,pt}} (\overline{\mathsf{Ideal}}, \bar{F})$. Eventually, using our parallel composition theorem (Thm. 2) with the $\mathsf{Dispatcher}$ automaton, we obtain the relation $(\overline{\mathsf{RealSigSys}}, \emptyset) \leq_{\mathsf{neg,pt}} (\overline{\mathsf{IdealSigSys}}, \bar{F})$, as needed.

7 Conclusion

We have introduced a new model for long-lived security protocols, based on task-PIOAs augmented with real-time task schedules. We express computational restrictions in terms of processing rates with respect to real time. The heart of our model is a long-term implementation relation, $\leq_{\mathsf{neg,pt}}$, which expresses security in any polynomial-length interval of time, despite of prior security violations. We have proved polynomial parallel composition and exponential sequential composition theorems for $\leq_{\mathsf{neg,pt}}$. Finally, we have applied the new theory to show security properties for a long-lived timestamping protocol.

This work suggests several directions for future work. First, for our particular timestamping case study, it remains to carry out the details of defining a higher-level abstract functionality specification for a long-lived timestamp service, and to use $\leq_{\mathsf{neg,pt}}$ to show that our ideal system, and hence, the real protocol, implements that specification.

We would also like to know whether or not it is possible to achieve stronger properties for long-lived timestamp services, such as reliably certifying a document for super-polynomial time.

It remains to use these definitions to study additional long-lived protocols and their security properties. The use of real time in the model should enable quantitative analysis of the rate of security degradation. Finally, it would be interesting to generalize the framework to allow the computational power of the various system components to increase with time.

References

1. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. In: Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC 1985), pp. 291–304 (1985)
2. Pfitzmann, B., Waidner, M.: A model for asynchronous reactive systems and its application to secure message transmission. In: IEEE Symposium on Security and Privacy, Oakland, CA, pp. 184–200. IEEE Computer Society, Los Alamitos (2001)

3. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Naor, M. (ed.) Proceedings of the 42nd Annual Symposium on Foundations of Computer Science, pp. 136–145. IEEE Computer Society, Los Alamitos (2001)
4. Goldreich, O.: Foundations of Cryptography: Basic Tools, vol. 1. Cambridge University Press, Cambridge (2001) (reprint of 2003)
5. Canetti, R., Cheung, L., Kaynar, D., Liskov, M., Lynch, N., Pereira, O., Segala, R.: Analyzing security protocols using time-bounded Task-PIOAs. Discrete Event Dynamic Systems 18(1), 111–159 (2008)
6. Ostrovsky, R., Yung, M.: How to withstand mobile virus attacks. In: Proceedings of 10th annual ACM Symposium on Principles of Distributed Computing (PODC 1991), pp. 51–59 (1991)
7. Anderson, R.: Two remarks on public key cryptology. Technical Report UCAM-CL-TR-549. University of Cambridge (2002)
8. Bellare, M., Miner, S.K.: A forward-secure digital signature scheme. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 431–448. Springer, Heidelberg (1999)
9. Canetti, R., Halevi, S., Katz, J.: A forward-secure public-key encryption scheme. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 255–271. Springer, Heidelberg (2003)
10. Bayer, D., Haber, S., Stornetta, S.W.: Improving the efficiency and reliability of digital time-stamping. In: Capocalli, R.M., Santis, A.D., Vaccaro, U. (eds.) Sequences II: Methods in Communication, Security, and Computer Science (Proceedings of the Sequences Workshop, 1991), pp. 329–334. Springer, Heidelberg (1993)
11. Haber, S.: Long-lived digital integrity using short-lived hash functions. Technical report, HP Laboratories (2006)
12. Haber, S., Kamat, P.: A content integrity service for long-term digital archives. In: Proceedings of the IS&T Archiving Conference (2006); Also published as Technical Memo HPL-2006-54, Trusted Systems Laboratory, HP Laboratories, Princeton
13. Mitchell, J., Ramanathan, A., Scedrov, A., Teague, V.: A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. Theoretical Computer Science 353, 118–164 (2006)
14. Backes, M., Pfitzmann, B., Waidner, M.: Secure asynchronous reactive systems. Cryptology ePrint Archive, Report 2004/082 (2004), http://eprint.iacr.org/
15. Müller-Quade, J., Unruh, D.: Long-term security and universal composability. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 41–60. Springer, Heidelberg (2007)
16. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic Journal of Computing 2(2), 250–273 (1995)
17. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
18. Merritt, M., Modugno, F., Tuttle, M.R.: Time constrained automata. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 408–423. Springer, Heidelberg (1991)
19. Canetti, R., Cheung, L., Kaynar, D., Lynch, N., Pereira, O.: Modeling bounded computation in long-lived systems. Cryptology ePrint Archive, Report 2007/406 (2007), http://eprint.iacr.org/
20. Canetti, R., Cheung, L., Kaynar, D., Lynch, N., Pereira, O.: Compositional security for Task-PIOAs. In: Sabelfeld, A. (ed.) 20th IEEE Computer Security Foundations Symposium, pp. 125–139. IEEE Computer Society Press, Los Alamitos (2007)

Contract-Directed Synthesis of Simple Orchestrators

Luca Padovani

Information Science and Technology Institute, University of Urbino
padovani@sti.uniurb.it

Abstract. The availability of repositories of Web service descriptions enables interesting forms of dynamic Web service discovery, such as searching for Web services exposing a specified behavior – or *contract*. This calls for a formal notion of contract equivalence satisfying two contrasting goals: being as coarse as possible so as to favor Web services reuse, and guaranteeing smooth client/service interaction. We study an equivalence relation under the assumption that interactions are controlled by *orchestrators*. We build a simple orchestration language on top of this theory and we show how to synthesize orchestrators out of Web services contracts.

1 Introduction

Web services are distributed processes equipped with a public description of their interface. Such description typically includes the type of messages exchanged with the service, the *operations* provided by the service [8], and also the behavior – or *contract* – supported by the service [1, 2]. The description is made public by registering the service in one or more Web service repositories [3, 10, 25] that can be queried and searched for finding services providing a particular contract. This calls for a formalization of the contract language and particularly of a *subcontract relation*, which is determined by comparing the sets of clients satisfied by different services.

In this work we express contracts using a fragment of CCS [12] with two choice operators ($+$ for external choice and $\oplus$ for internal choice) without relabeling, restriction, and parallel composition. For instance, the contract $\sigma = a.c.(\overline{b} \oplus \overline{d})$ describes a service that accepts two messages a and c (in this order) and then decides internally whether to send back either $\overline{b}$ or $\overline{d}$. The contract $\rho = \overline{a}.\overline{c}.(b.\mathsf{e} + d.\mathsf{e})$ describes a client that sends two messages a and c (in this order), then waits for either the message b or the message d, and finally terminates (e denotes successful termination). The compliance relation $\rho \dashv \sigma$ tells us that the client ρ is satisfied by the service σ, because every possible interaction between ρ and σ leads to the client terminating successfully. This is not true for ρ and $\sigma' = a.c.(\overline{b} \oplus \overline{c})$, because the service with contract σ' may internally decide to send a message $\overline{c}$ that the client is not willing to accept, hence $\rho \not\dashv \sigma'$. The subcontract relation $\sigma \preceq \tau$, where $\tau = a.c.\overline{b}$, tells us that every client satisfied by σ (including ρ) is also satisfied by τ. This is because τ is more deterministic than σ.

Formal notions of compliance and subcontract relation may be used for implementing contract-based query engines. The query for services that satisfy ρ is answered with the set $\{\sigma \mid \rho \dashv \sigma\}$. The complexity of running this query grows with the number of services stored in the repository. A better strategy is to compute the *dual contract* of ρ,

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 131–146, 2008.

denoted by $\rho^\perp$, which represents the canonical service satisfying ρ ($\rho \dashv \rho^\perp$) and then answering the query with the set $\{\sigma \mid \rho^\perp \preceq \sigma\}$. If $\rho^\perp$ is the $\preceq$-smallest service that satisfies ρ, we are guaranteed that no service is mistakenly excluded. The advantage of this approach is that $\preceq$ can be precomputed when services are registered in the repository, and the query engine needs only scan through the $\preceq$-minimal contracts.

When looking for a suitable theory defining $\dashv$ and $\preceq$, the testing framework [11, 16] and the *must* preorder seem particularly appealing: clients are tests, compliance encodes the passing of a test, and the subcontract relation is the liveness-preserving preorder induced by the compliance relation. Unfortunately, the *must* preorder excludes many relations that are desirable in the context of Web services. For example, a service with contract $a+b$ cannot replace a service with contract a, despite the fact that $a+b$ offers more options a. The reason is that the client $\rho' = \overline{a}.\mathsf{e} + \overline{b}.c.\mathsf{e}$ complies with a simply because no interaction on b is possible, whereas it can get stuck when interacting with $a+b$ because such service does not offer $\overline{c}$ after b. As another example, the client $\rho'' = \overline{c}.\overline{a}.(b.\mathsf{e} + d.\mathsf{e})$ fails to interact successfully with σ above because it sends the messages $\overline{a}$ and $\overline{c}$ in the wrong order.

In this work we deviate from the classical testing framework by making client and service interact under the supervision of an *orchestrator*. In the Web services domain, an orchestrator coordinates in a centralized way two (or more) interacting parties so as to achieve a specific goal, in our case to guarantee client satisfaction. The orchestrator cannot affect the internal decisions of client and service, but it can affect the way they try to synchronize with each other. In our framework an orchestrator is a *bounded, directional, controlled buffer*: the buffer is bounded in that it can store a finite amount of messages; the buffer is directional in that it distinguishes messages sent to the client from messages sent to the service; the buffer is controlled by *orchestration actions*:

- An asynchronous action $\langle a, \varepsilon \rangle$ indicates that the orchestrator accepts a message a from the client, without delivering it to the service; dually, $\langle \overline{a}, \varepsilon \rangle$ indicates that the orchestrator sends a message $\overline{a}$ (previously received from the service) to the client;
- an action of the form $\langle \varepsilon, \alpha \rangle$ indicates a similar capability on the service side;
- a synchronous action $\langle a, \overline{a} \rangle$ indicates that the orchestrator accepts a message a from the client, provided that the service can receive $\overline{a}$; dually for $\langle \overline{a}, a \rangle$.

The orchestrator $f = \langle a, \overline{a} \rangle$ makes the client ρ' above compliant with $a+b$, because it forbids any interaction on b; the orchestrator $g = \langle c, \varepsilon \rangle.\langle a, \varepsilon \rangle.\langle \varepsilon, \overline{a} \rangle.\langle \varepsilon, \overline{c} \rangle.(\langle \overline{b}, b \rangle + \langle \overline{d}, d \rangle)$ makes the client ρ'' above compliant with σ, because the orchestrator accepts c followed by a from the client, and then delivers them in the order expected by the service. Orchestrators can be interpreted as morphisms transforming service contracts: the relation $f : a \preceq a+b$ states that every client satisfied by a is also satisfied by $a+b$ by means of the orchestrator f; the relation $g : c.a.(\overline{b} \oplus \overline{d}) \preceq a.c.(\overline{b} \oplus \overline{d})$ states that every client that sends $\overline{c}$ before $\overline{a}$ and then waits for either $\overline{b}$ or $\overline{d}$ can also be satisfied by $a.c.(\overline{b} \oplus \overline{d})$, provided that g orchestrates its interaction with the service. On the other hand, no orchestrator is able to make ρ interact successfully with σ', because the internal decisions taken by σ' cannot be controlled.

Structure of the paper. In §2 we define contracts and we recast the standard testing framework in our setting. In §3 we define compliance and subcontract relations for

orchestrated processes. From such definitions we design a simple orchestration language and prove its main properties. §4 shows how to compute the dual contract and §5 presents an algorithm for synthesizing orchestrators by comparing service contracts. In §6 we show the algorithm at work on two less trivial examples. We conclude in §7 with a discussion of the main achievements of this work and possible directions for future research. Proofs can be found in the full version [23].

Related work. This work originated by revisiting CCS without τ's [12] in the context of Web services. Early attempts to define a reasonable subcontract relation [5] have eventually led to the conclusion that some control over actions is necessary: [21] proposes a static form of control that makes use of explicit contract interfaces; [6] proposes a dynamic form of action filtering. The present work elaborates on the idea of [6] by adding asynchrony and buffering: this apparently simple addition significantly increases the technicalities of the resulting theory. The subcontract relation presented in this work, because of its liveness-preserving property, has connections with and extends the subtyping relation on session types [15, 17] and stuck-free conformance relation [14].

WS-BPEL [1] is often presented as an orchestration language for Web services. Remarkably WS-BPEL features boil down to storing incoming messages into variables (buffering) and controlling the interactions of other parties. Our orchestrators can be seen as streamlined WS-BPEL orchestrators in which all the internal nondeterminism of the orchestrator itself is abstracted away. ORC [22] is perhaps the most notable example of orchestration-oriented, algebraic language. The peculiar operators $\gg$ and **where** of ORC represent different forms of *pipelining* and can be seen as orchestration actions in conjunction with the composition operator $\cdot$ of orchestrators (§3).

In software architectures there has been extensive research on the automatic synthesis of connectors for software components (see for example [18]) and attempts have been made to apply the resulting approaches to Web services [19]. In these works the problem is to connect a set of tightly coupled components so as to guarantee some safety properties among which deadlock freeness. Unlike Web services, where communication is peer-to-peer, architectural topologies can be arbitrarily complex. This leads to the generation of connectors that only work for specific architectural configurations. In our approach orchestrators are "proofs" (in the Curry-Howard sense) for $\preceq$ whose transitivity stems directly from the ability of composing orchestrators incrementally.

2 Contracts

The syntax of contracts makes use of a denumerable set $\mathcal{N}$ of *names* ranged over by $a, b, \ldots$ and of a denumerable set of *variables* ranged over by $x, y, \ldots$; we write $\overline{\mathcal{N}}$ for the set of *co-names* $\overline{a}$, where $a \in \mathcal{N}$. Names represent input actions, while co-names represent output actions; we let $\alpha, \beta, \ldots$ range over actions; we let $\varphi, \varphi', \ldots$ range over sequences of actions; we let $\mathrm{R}, \mathrm{S}, \ldots$ range over finite sets of actions; we let $\overline{\overline{\alpha}} = \alpha$ and $\overline{\mathrm{R}} = \{\overline{\alpha} \mid \alpha \in \mathrm{R}\}$. The meaning of names is left unspecified: they can stand for ports, operations, message types, and so forth. Contracts are ranged over by $\rho, \sigma, \tau, \ldots$ and their syntax is given by the following grammar:

$$\sigma ::= 0 \mid \alpha.\sigma \mid \sigma + \sigma \mid \sigma \oplus \sigma \mid \mathtt{rec}\, x.\sigma \mid x$$

The notions of free and bound variables in contracts are standard, being $\mathtt{rec}\ x$ the only binder. In the following we write $\sigma\{^{\tau}/_{x}\}$ for the contract σ' that is the same as σ except that every free occurrence of x has been replaced by τ. We assume variables to be *guarded*: every free occurrence of x in a term $\mathtt{rec}\ x.\sigma$ must be found in a subterm of σ having the form $\alpha.\sigma'$. The null contract $\mathtt{0}$ describes the idle process that offers no action (we will omit trailing $\mathtt{0}$'s); the contract $\alpha.\sigma$ describes a process that offers the action α and then behaves as σ; the contract $\sigma + \tau$ is the *external choice* of σ and τ and describes a process that can either behave as σ or as τ depending on the party it is interacting with; the contract $\sigma \oplus \tau$ is the *internal choice* of σ and τ and describes a process that autonomously decides to behave as either σ or τ; the contract $\mathtt{rec}\ x.\sigma$ describes a process that behaves as $\sigma\{^{\mathtt{rec}\ x.\sigma}/_{x}\}$.

The transition relation of contracts is inductively defined by the following rules (symmetric rules for $+$ and $\oplus$ are omitted):

$$\alpha.\sigma \xrightarrow{\alpha} \sigma \qquad \sigma \oplus \tau \longrightarrow \sigma \qquad \mathtt{rec}\ x.\sigma \longrightarrow \sigma\{^{\mathtt{rec}\ x.\sigma}/_{x}\}$$

$$\frac{\sigma \longrightarrow \sigma'}{\sigma + \tau \longrightarrow \sigma' + \tau} \qquad \frac{\sigma \xrightarrow{\alpha} \sigma'}{\sigma + \tau \xrightarrow{\alpha} \sigma'}$$

The relation $\longrightarrow$ denotes internal transitions, while $\xrightarrow{\alpha}$ denotes external transitions labeled with an action α. Overall the transition relation is the same as that of CCS without τ's [12]. In particular, the fact that $+$ stands for an external choice is clear from the fourth rule, where the internal transition $\sigma \longrightarrow \sigma'$ does not preempt the τ branch. The guardedness assumption we made earlier ensures that the number of consecutive internal transitions in any derivation of a contract is finite (*strong convergence*). We write $\Longrightarrow$ for the reflexive, transitive closure of $\longrightarrow$; let $\overset{\alpha}{\Longrightarrow}$ be $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$; we write $\sigma \xrightarrow{\alpha}$ if there exists σ' such that $\sigma \xrightarrow{\alpha} \sigma'$, and similarly for $\sigma \overset{\alpha}{\Longrightarrow}$; let $\mathtt{init}(\sigma) \stackrel{\text{def}}{=} \{\alpha \mid \sigma \overset{\alpha}{\Longrightarrow}\}$.

The transition relation above describes the transitions of a contract from the point of view of the process exposing the contract. The notion of *contract continuation*, which we are to define next, considers the point of view of the process it is interacting with.

Definition 1 (contract continuation). *Let* $\sigma \overset{\alpha}{\Longrightarrow}$. *The* continuation *of* σ *with respect to* α, *notation* $\sigma(\alpha)$, *is defined as* $\sigma(\alpha) \stackrel{\text{def}}{=} \bigoplus_{\sigma \Longrightarrow \xrightarrow{\alpha} \sigma'} \sigma'$. *We generalize the notion of continuation to finite sequences of actions so that* $\sigma(\varepsilon) = \sigma$ *and* $\sigma(\alpha\varphi) = \sigma(\alpha)(\varphi)$.

For example, $a.b \oplus a.c \overset{a}{\Longrightarrow} b$ (the process knows which branch has been taken) but $(a.b \oplus a.c)(a) = b \oplus c$ (the party interacting with $a.b \oplus a.c$ does not know which branch has been taken after seeing an a action, hence it considers both). Because of the guardedness condition there is a finite number of residuals σ' such that $\sigma \Longrightarrow \xrightarrow{\alpha} \sigma'$, hence $\sigma(\alpha)$ is well defined. Moreover, the set $D(\sigma) = \{\sigma(\varphi) \mid \sigma \overset{\varphi}{\Longrightarrow}\}$ is always finite (see [23]). This is a consequence of the fact that our contracts are finite representations of regular trees, which have a finite number of different subtrees. We will exploit this property throughout the paper for defining functions over contracts by induction on the set $D(\sigma)$ above.

The *ready sets* of a contract tell us about its internal nondeterminism. We say that σ has *ready set* R, written $\sigma \Downarrow \mathrm{R}$, if $\sigma \Longrightarrow \sigma'$ and $\mathrm{R} = \mathtt{init}(\sigma')$. Intuitively, $\sigma \Downarrow \mathrm{R}$ means that σ can independently evolve, by means of internal transitions, to another contract σ' which only offers the actions in R. For example, $\{a,b\}$ is the only ready set of $a+b$ (both a and b are always available), whereas the ready sets of $a \oplus b$ are $\{a\}$, $\{b\}$, and $\{a,b\}$ (the contract $a \oplus b$ may evolve into a state where only a is available, or only b is available, or both a and b are available).

As in the classical testing framework we model client satisfaction by means of a special action e. A client contract ρ is *compliant with* a service contract σ if every maximal, finite interaction of ρ and σ leads to a residual client contract ρ' such that $\rho' \xrightarrow{\mathtt{e}}$. Here we provide an equivalent coinductive definition which relates more directly with its weak variant in §3.

Definition 2 (strong compliance). *We say that $\mathscr{C}$ is a* strong compliance relation *if $(\rho,\sigma) \in \mathscr{C}$ implies that*

1. *$\rho \Downarrow \mathrm{R}$ and $\sigma \Downarrow \mathrm{S}$ implies either $\mathtt{e} \in \mathrm{R}$ or $\overline{\mathrm{R}} \cap \mathrm{S} \neq \emptyset$, and*
2. *$\rho \overset{\overline{\alpha}}{\Longrightarrow}$ and $\sigma \overset{\alpha}{\Longrightarrow}$ implies $(\rho(\overline{\alpha}), \sigma(\alpha)) \in \mathscr{C}$.*

We write $\dashv$ for the largest strong compliance relation.

Condition (1) requires that for every combination of states R and S of the client and of the service, either the client has terminated successfully ($\mathtt{e} \in \mathrm{R}$) or the client and the service can synchronize (there is an action $\alpha \in \mathrm{S}$ such that $\overline{\alpha} \in \overline{\mathrm{R}}$). For instance $a+b \dashv \overline{a} \oplus \overline{b}$ and $a \oplus b \dashv \overline{a}+\overline{b}$, but $a \oplus b \not\dashv \overline{a} \oplus \overline{b}$. Condition (2) ensures that every synchronization produces a residual client that is compliant with the residual service. For instance $\overline{a}.(b \oplus d) \not\dashv a.\overline{b}+a.\overline{d}$ because after the synchronization on a this reduces to $b \oplus d \not\dashv \overline{b} \oplus \overline{d}$.

The (strong) compliance relation provides us with the most natural equivalence for comparing services: the (service) contract σ is "smaller than" the (service) contract τ if every client that is compliant with σ is also compliant with τ.

Definition 3 (strong subcontract). *We say that σ is a* strong subcontract *of τ, notation $\sigma \sqsubseteq \tau$, if for every ρ we have $\rho \dashv \sigma$ implies $\rho \dashv \tau$. We write $\simeq$ for the equivalence relation induced by $\sqsubseteq$, that is $\simeq = \sqsubseteq \cap \sqsupseteq$.*

For instance, we have $a \oplus b \sqsubseteq a$ because every client that is satisfied to interact with a service that may decide to offer either a or b is also satisfied by a service that systematically offers a. On the other hand $a.(b+c) \not\sqsubseteq a.b+a.c$ because after interacting on a, a client of $a.(b+c)$ can decide whether to interact on b or on c, whereas in $a.b+a.c$ only one of these actions is available, according to the branch taken by the service. In general the set-theoretic definition of the preorder above is rather difficult to work with directly, and an alternative characterization such as the following is preferred.

Definition 4 (coinductive strong subcontract). *We say that $\mathscr{S}$ is a* coinductive strong subcontract relation *if $(\sigma,\tau) \in \mathscr{S}$ implies*

1. *$\tau \Downarrow \mathrm{S}$ implies $\sigma \Downarrow \mathrm{R}$ and $\mathrm{R} \subseteq \mathrm{S}$ for some R, and*
2. *$\tau \overset{\alpha}{\Longrightarrow}$ implies $\sigma \overset{\alpha}{\Longrightarrow}$ and $(\sigma(\alpha), \tau(\alpha)) \in \mathscr{S}$.*

Condition (1) requires τ to be more deterministic than σ (every ready set of τ has a corresponding one of σ that offers fewer actions). Condition (2) requires τ to offer no more actions than those offered by σ, and every continuation after an action offered by both σ and τ to be in the subcontract relation. We conclude this section with the most important properties enjoyed by $\sqsubseteq$.

Proposition 1. *The following properties hold:*

1. *$\sqsubseteq$ is the largest coinductive subcontract relation;*
2. *$\sqsubseteq$ coincides with the* must *preorder [11, 12, 16] for strongly convergent processes;*
3. *$\sqsubseteq$ is a precongruence with respect to all the operators of the contract language.*

Property (1) states the correctness of Definition 4 as an alternative characterization for $\sqsubseteq$. Property (2) connects $\sqsubseteq$ with the well-known *must* testing preorder. This result is not entirely obvious because the notion of "passing a test" we use differs from that used in the standard testing framework (see [21] for more details). Finally, property (3) states that $\sqsubseteq$ is well behaved and that it can be used for modular refinement. The weak variant of the subcontract relation that we will define in §3 does not enjoy this property in general, but not without reason as we will see.

3 Simple Orchestrators

The strong compliance relation requires that progress must always be guaranteed for *both* client and service unless the client is satisfied. We relax this requirement and assume that an orchestrator mediates the interaction of a client and a service by ensuring that progress is guaranteed for *at least one* of the parties. The orchestrator must be *fair*, in the sense that client and service must have equal opportunities to make progress.

Weak compliance and subcontract relations. Orchestrators perform actions having one of the following forms: the action $\langle \alpha, \varepsilon \rangle$ means that the orchestrator offers α to the client; the action $\langle \varepsilon, \overline{\alpha} \rangle$ means that the orchestrator offers $\overline{\alpha}$ to the service; the action $\langle \alpha, \overline{\alpha} \rangle$ means that the orchestrator simultaneously offers α to the client and $\overline{\alpha}$ to the service; we let $\mu, \mu', \ldots$ range over orchestration actions and $\mathrm{A}, \mathrm{A}', \ldots$ range over sets of orchestrator actions. A buffer is a map $\{\circ, \bullet\} \times \overline{\mathscr{N}} \to \mathbb{Z}$ associating pairs $(r, \overline{a})$ with the number of $\overline{a}$ messages stored in the buffer and available for delivery to the role r, where r can be $\circ$ for "client" or $\bullet$ for "service"; we let $\mathbb{B}, \mathbb{B}', \ldots$ range over buffers. For technical reasons we allow $\mathtt{cod}(\mathbb{B})$ – the codomain of $\mathbb{B}$ – to range over $\mathbb{Z}$, although every well-formed buffer will always contain a nonnegative number of messages. We write $\tilde{\emptyset}$ for the empty buffer, the one having $\{0\}$ as codomain; we write $\mathbb{B}[(r, \overline{a}) \mapsto n]$ for the buffer $\mathbb{B}'$ which is the same as $\mathbb{B}$ except that $(r, \overline{a})$ is associated with n; we write $\mathbb{B}\mu$ for the buffer $\mathbb{B}$ updated after the action μ:

$$
\begin{array}{rll}
\mathbb{B}\langle a, \varepsilon \rangle = \mathbb{B}[(\bullet, \overline{a}) \mapsto \mathbb{B}(\bullet, \overline{a}) + 1] & & \text{(accept } \overline{a} \text{ from the client)} \\
\mathbb{B}\langle \overline{a}, \varepsilon \rangle = \mathbb{B}[(\circ, \overline{a}) \mapsto \mathbb{B}(\circ, \overline{a}) - 1] & & \text{(send } \overline{a} \text{ to the client)} \\
\mathbb{B}\langle \varepsilon, a \rangle = \mathbb{B}[(\circ, \overline{a}) \mapsto \mathbb{B}(\circ, \overline{a}) + 1] & & \text{(accept } \overline{a} \text{ from the service)} \\
\mathbb{B}\langle \varepsilon, \overline{a} \rangle = \mathbb{B}[(\bullet, \overline{a}) \mapsto \mathbb{B}(\bullet, \overline{a}) - 1] & & \text{(send } \overline{a} \text{ to the service)} \\
\mathbb{B}\langle \alpha, \overline{\alpha} \rangle = \mathbb{B} & & \text{(synchronize client and service)}
\end{array}
$$

We say that $\mathbb{B}$ has rank k, or is a k-buffer, if $\mathtt{cod}(\mathbb{B}) \subseteq [0,k]$; we say that the k-buffer $\mathbb{B}$ *enables* the orchestration action μ, notation $\mathbb{B} \vdash_k \mu$, if $\mathbb{B}\mu$ is still a k-buffer. For instance $\tilde{\emptyset} \vdash_1 \langle a, \varepsilon \rangle$ but $\tilde{\emptyset} \nvdash_k \langle \overline{a}, \varepsilon \rangle$ because $-1 \in \mathtt{cod}(\tilde{\emptyset}\langle \overline{a}, \varepsilon \rangle)$. We extend the notion to sets of actions so that $\mathbb{B} \vdash_k \mathrm{A}$ if $\mathbb{B}$ enables every action in A. Synchronization actions are enabled regardless of the rank of the buffer, because they leave the buffer unchanged.

When an orchestrator mediates the interaction between a client and a service, it proposes at each interaction step a set of orchestration actions A. If R is a client ready set and S is a service ready set, then $\mathrm{A} \circ \mathrm{S}$ denotes the service ready set perceived by the client and $\mathrm{R} \bullet \mathrm{A}$ denotes the client ready set perceived by the service:

$$\mathrm{A} \circ \mathrm{S} \stackrel{\text{def}}{=} \{\alpha \mid \langle \alpha, \varepsilon \rangle \in \mathrm{A}\} \cup \{\alpha \in \mathrm{S} \mid \langle \alpha, \overline{\alpha} \rangle \in \mathrm{A}\}$$
$$\mathrm{R} \bullet \mathrm{A} \stackrel{\text{def}}{=} \{\overline{\alpha} \mid \langle \varepsilon, \overline{\alpha} \rangle \in \mathrm{A}\} \cup \{\overline{\alpha} \in \mathrm{R} \mid \langle \alpha, \overline{\alpha} \rangle \in \mathrm{A}\}$$

Namely, the client sees an action α if either that action is provided asynchronously by the orchestrator ($\langle \alpha, \varepsilon \rangle \in \mathrm{A}$), or if it is provided by the service ($\alpha \in \mathrm{S}$) and the orchestrator does not hide it ($\langle \alpha, \overline{\alpha} \rangle \in \mathrm{A}$); symmetrically for service. We now possess all the technical notions for defining the compliance relation with orchestrators.

Definition 5 (weak compliance). *We say that $\mathscr{D}_k$ is a* coinductive weak k-compliance relation *if $(\mathbb{B}, \rho, \sigma) \in \mathscr{D}_k$ implies that $\mathbb{B}$ is a k-buffer and there exists a set of orchestration actions A such that $\mathbb{B} \vdash_k \mathrm{A}$ and*

1. *$\rho \Downarrow \mathrm{R}$ and $\sigma \Downarrow \mathrm{S}$ implies either $\mathtt{e} \in \mathrm{R}$ or $\overline{\mathrm{R}} \cap (\mathrm{A} \circ \mathrm{S}) \neq \emptyset$ or $(\mathrm{R} \bullet \mathrm{A}) \cap \overline{\mathrm{S}} \neq \emptyset$, and*
2. *$\rho \stackrel{\overline{\varphi}}{\Longrightarrow}$ and $\sigma \stackrel{\varphi'}{\Longrightarrow}$ and $\langle \varphi, \overline{\varphi}' \rangle \in \mathrm{A}$ implies $(\mathbb{B}\langle \varphi, \overline{\varphi}' \rangle, \rho(\overline{\varphi}), \sigma(\varphi')) \in \mathscr{D}_k$.*

We write $\rho \dashv\!|_k \sigma$ if there exists $\mathscr{D}_k$ such that $(\tilde{\emptyset}, \rho, \sigma) \in \mathscr{D}_k$.

While commenting on the definition of weak compliance, it is useful to compare it with Definition 2. A tuple $(\mathbb{B}, \rho, \sigma)$ represents the state of the system, which comprises the client ρ, the service σ, and the buffer $\mathbb{B}$. The definition requires the existence of a set A of orchestration actions compatible with the state of the buffer ($\mathbb{B} \vdash_k \mathrm{A}$) so that: (1) for every combination of client states R and service states S, either the client is satisfied ($\mathtt{e} \in \mathrm{R}$) or progress is guaranteed for the client ($\overline{\mathrm{R}} \cap (\mathrm{A} \circ \mathrm{S}) \neq \emptyset$) or it is guaranteed for the service ($(\mathrm{R} \bullet \mathrm{A}) \cap \overline{\mathrm{S}} \neq \emptyset$); (2) whatever action is executed, the state of the system after the action is still in the compliance relation. For example, if $\rho \stackrel{a}{\Longrightarrow}$ and $\langle \overline{a}, \varepsilon \rangle \in \mathrm{A}$, then $(\mathbb{B}\langle \overline{a}, \varepsilon \rangle, \rho(a), \sigma)$ must be in the compliance relation.

Directionality of the buffer is necessary for preserving the correct flow of messages between client and service and also for fairness: it prevents the orchestrator from satisfying one of the parties by sending back its own messages. The index k prevents the orchestrator from accepting an unlimited number of messages from just one of the parties, letting the other one starve for an interaction.

Weak compliance induces the weak subcontract relation as follows:

Definition 6 (weak subcontract). *We say that σ is a* weak subcontract *of τ, notation $\sigma \preceq \tau$, if there exists k such that $\rho \dashv \sigma$ implies $\rho \dashv\!|_k \tau$ for every ρ.*

Namely, when $\sigma \preceq \tau$ a service with contract τ can replace a service with contract σ because every client satisfied by σ ($\rho \dashv \sigma$) can also be satisfied by τ ($\rho \dashv\!|_k \tau$) by means

of some orchestrator. The weak subcontract includes the strong one: when proving $\rho \dashv\!\vert_k \tau$ just consider the set $\mathrm{A} = \{\langle \alpha, \overline{\alpha} \rangle \mid \tau \xRightarrow{\alpha}\}$. On the other hand, we have $a \preceq a + b$ (by filtering b out), $a.\beta.\sigma \preceq \beta.a.\sigma$ (by delaying $\overline{a}$ from the client until the service needs it), and $\alpha.\overline{b}.\sigma \preceq \overline{b}.\alpha.\sigma$ (by delaying $\overline{b}$ from the service until the client needs it).

As usual the set-theoretic definition of subcontract relation is not particularly enlightening, hence the following alternative characterization.

Definition 7 (coinductive weak subcontract). *We say that $\mathscr{W}_k$ is a* coinductive weak k-subcontract relation *if $(\mathbb{B}, \sigma, \tau) \in \mathscr{W}_k$ implies that $\mathbb{B}$ is a k-buffer and there exists a set of orchestration actions A such that $\mathbb{B} \vdash_k \mathrm{A}$ and*

1. *$\tau \Downarrow \mathrm{S}$ implies either $(\sigma \Downarrow \mathrm{R}$ and $\mathrm{R} \subseteq \mathrm{A} \circ \mathrm{S}$ for some $\mathrm{R})$ or $(\emptyset \bullet \mathrm{A}) \cap \overline{\mathrm{S}} \neq \emptyset$, and*
2. *$\tau \xRightarrow{\varphi'}$ and $\langle \varphi, \overline{\varphi}' \rangle \in \mathrm{A}$ implies $\sigma \xRightarrow{\varphi}$ and $(\mathbb{B}\langle \varphi, \overline{\varphi}' \rangle, \sigma(\varphi), \tau(\varphi')) \in \mathscr{W}_k$.*

We write $\sigma \preceq^{\mathsf{c}} \tau$ if there exists $\mathscr{W}_k$ such that $(\tilde{\emptyset}, \sigma, \tau) \in \mathscr{W}_k$.

Condition (1) requires that either τ can be made more deterministic than σ by means of the orchestrator (the ready set $\mathrm{A} \circ \mathrm{S}$ of the orchestrated service has a corresponding one of σ that offers fewer actions), or that τ can be satisfied by the orchestrator without any help from the client ($(\emptyset \bullet \mathrm{A}) \cap \overline{\mathrm{S}} \neq \emptyset$ implies that $\langle \varepsilon, \overline{\alpha} \rangle \in \mathrm{A}$ and $\alpha \in \mathrm{S}$ for some α). Condition (2) poses the usual requirement that the continuations must be in the subcontract relation. The two definitions of weak subcontract are equivalent:

Theorem 1. $\preceq = \preceq^{\mathsf{c}}$.

Remark 1. Theorem 1 entails a nontrivial property of $\preceq$ that makes $\preceq$ suitable as a subcontract relation: $\sigma \preceq \tau$ means that every client ρ satisfied by σ is weakly compliant with τ by means of *some* orchestrator which, in principle, may depend on ρ. On the other hand, $\sigma \preceq^{\mathsf{c}} \tau$ means that there exists an orchestrator such that every client satisfied by σ is weakly compliant with τ by means of *that one* orchestrator. In practical terms, this allows us to precompute not only the subcontract relation $\preceq$ but also the orchestrator that proves it, regardless of the client performing the query.

A simple orchestration language. The definition of weak compliance relation suggests a representation of orchestrators as algebraic terms specifying sets of orchestration actions along with corresponding continuations. Following this intuition we propose a language of simple orchestrators:

$$f ::= \mathtt{0} \mid \mu.\sigma \mid f + f \mid \mathtt{rec}\, x.f \mid x$$

We let $f, g, h, \ldots$ range over orchestrators. The orchestrator $\mathtt{0}$ offers no action (we will omit trailing $\mathtt{0}$'s); the orchestrator $\mu.f$ offers the action μ and then continues as f; the orchestrator $f + g$ offers the actions offered by either f or g; recursive orchestrators can be expressed by means of $\mathtt{rec}\ x.f$ and recursion variables in the usual way. As for contracts, we make the assumption that recursion variables must be guarded by at least one orchestration action. Orchestrators do not exhibit internal nondeterminism. This calls for a transition relation merely expressing which orchestration actions are

available. To this aim, we first define a predicate $f \overset{\mu}{\not\longmapsto}$ meaning that f *cannot* perform the orchestration action μ:

$$\mathsf{0} \overset{\mu}{\not\longmapsto} \qquad \frac{\mu \neq \mu'}{\mu'.f \overset{\mu}{\not\longmapsto}} \qquad \frac{f \overset{\mu}{\not\longmapsto} \quad g \overset{\mu}{\not\longmapsto}}{f+g \overset{\mu}{\not\longmapsto}} \qquad \frac{f\{\mathtt{rec}\, x.f/_x\} \overset{\mu}{\not\longmapsto}}{\mathtt{rec}\, x.f \overset{\mu}{\not\longmapsto}}$$

The transition relation of orchestrators is the least relation inductively defined by the following rules (symmetric rule for $+$ is omitted):

$$\mu.f \overset{\mu}{\longmapsto} f \qquad \frac{f \overset{\mu}{\longmapsto} f' \quad g \overset{\mu}{\longmapsto} g'}{f+g \overset{\mu}{\longmapsto} f'+g'} \qquad \frac{f \overset{\mu}{\longmapsto} f' \quad g \overset{\mu}{\not\longmapsto}}{f+g \overset{\mu}{\longmapsto} f'} \qquad \frac{f\{\mathtt{rec}\, x.f/_x\} \overset{\mu}{\longmapsto} f'}{\mathtt{rec}\, x.f \overset{\mu}{\longmapsto} f'}$$

Note that $f \overset{\mu}{\longmapsto} f'$ and $f \overset{\mu}{\longmapsto} f''$ implies $f' = f''$. We write $f \overset{\mu}{\longmapsto}$ if there exists f' such that $f \overset{\mu}{\longmapsto} f'$; we write $f \overset{\mu_1 \cdots \mu_n}{\longmapsto}$ if $f \overset{\mu_1}{\longmapsto} \cdots \overset{\mu_n}{\longmapsto}$. Let $\mathtt{init}(f) \overset{\text{def}}{=} \{\mu \mid f \overset{\mu}{\longmapsto}\}$. We say that f is a valid orchestrator of rank k, or is a k-orchestrator, if $f \overset{\mu_1 \cdots \mu_n}{\longmapsto}$ implies that $\tilde{\emptyset}\mu_1 \cdots \mu_n$ is a k-buffer. Not every term f denotes a valid orchestrator of finite rank. For instance $\mathtt{rec}\, x.\langle a, \varepsilon\rangle.x$ is invalid because it accepts an unbounded number of messages from the client; $\langle \overline{a}, \varepsilon\rangle$ is invalid because it tries to deliver a message that it has not received; $\langle \varepsilon, a\rangle.\langle \overline{a}, \varepsilon\rangle$ is a valid orchestrator of rank 1 (or greater). In the following we will always work with valid orchestrators of finite rank.

When an orchestrator f mediates an interaction, it is as if the service operates while being filtered by f. The dual point of view, in which f filters the client, is legitimate, but it does not allow us to study the theory of simple orchestrators in a client-independent way, as by Remark 1. We extend syntax and semantics of contracts with terms of the form $f \cdot \sigma$, representing the application of the orchestrator f to the contract σ:

$$\frac{\sigma \longrightarrow \sigma'}{f \cdot \sigma \longrightarrow f \cdot \sigma'} \qquad \frac{f \overset{\langle \varepsilon, \overline{\alpha}\rangle}{\longmapsto} f' \quad \sigma \overset{\alpha}{\longrightarrow} \sigma'}{f \cdot \sigma \longrightarrow f' \cdot \sigma'} \qquad \frac{f \overset{\langle \alpha, \overline{\alpha}\rangle}{\longmapsto} f' \quad \sigma \overset{\alpha}{\longrightarrow} \sigma'}{f \cdot \sigma \overset{\alpha}{\longrightarrow} f' \cdot \sigma'} \qquad \frac{f \overset{\langle \alpha, \varepsilon\rangle}{\longmapsto} f'}{f \cdot \sigma \overset{\alpha}{\longrightarrow} f' \cdot \sigma}$$

In the first rule the service performs internal actions regardless of the orchestrator; in the second and third rules the service interacts with the orchestrator, and possibly with the client if the orchestrator allows it; in the last rule the orchestrator offers its asynchronous actions to the client independently of the service.

The orchestration language we have just devised is correct and complete with respect to the weak subcontract relation, in the following sense:

Theorem 2. *$\sigma \preceq \tau$ if and only if $\sigma \sqsubseteq f \cdot \tau$ for some k-orchestrator f.*

Orchestrators as morphisms. According to Theorem 2, orchestrators act as functions from contracts to contracts: if $\sigma \preceq \tau$, then there is a function f mapping services with contract τ into services with contract $f \cdot \tau$ such that $\sigma \sqsubseteq f \cdot \tau$. The function determined by an orchestrator can be effectively computed as follows:

$$f(\sigma) \overset{\text{def}}{=} \bigoplus_{\sigma \Downarrow \mathrm{R}} \begin{cases} \sum_{f \overset{\langle \alpha, \varepsilon\rangle}{\longmapsto} f'} \alpha.f'(\sigma) + \sum_{f \overset{\langle \alpha, \overline{\alpha}\rangle}{\longmapsto} f', \alpha \in \mathrm{R}} \alpha.f'(\sigma(\alpha)) & \text{if } (\emptyset \bullet \mathtt{init}(f)) \cap \overline{\mathrm{R}} = \emptyset \\ \Big(\big(\sum_{f \overset{\langle \alpha, \varepsilon\rangle}{\longmapsto} f'} \alpha.f'(\sigma) + \sum_{f \overset{\langle \alpha, \overline{\alpha}\rangle}{\longmapsto} f', \alpha \in \mathrm{R}} \alpha.f'(\sigma(\alpha))\big) \oplus \mathsf{0}\Big) & \\ \quad + \bigoplus_{f \overset{\langle \varepsilon, \overline{\alpha}\rangle}{\longmapsto} f', \alpha \in \mathrm{R}} f'(\sigma(\alpha)) & \text{otherwise} \end{cases}$$

For example consider $f \stackrel{\text{def}}{=} \langle a,\varepsilon\rangle.\langle c,\varepsilon\rangle.(\langle\varepsilon,\overline{a}\rangle.\langle\overline{b},b\rangle + \langle\varepsilon,\overline{c}\rangle.\langle\overline{d},d\rangle)$. Then we have $f(a.\overline{b}) = a.c.\overline{b}$; $f(a.\overline{b}+c.\overline{d}) = a.c.(\overline{b}\oplus\overline{d})$; $f(a.\overline{b}\oplus c.\overline{d}) = a.c.(0\oplus\overline{b}\oplus\overline{d})$. In general we have $\langle\alpha,\overline{\alpha}\rangle.f(\alpha.\sigma) = \alpha.f(\sigma)$ and $\langle\alpha,\varepsilon\rangle.f(\sigma) = \alpha.f(\sigma)$ and $\langle\varepsilon,\overline{\alpha}\rangle.f(\alpha.\sigma) = f(\sigma)$. The next result proves that $f(\sigma)$ is indeed the contract of the orchestrated service $f\cdot\sigma$:

Theorem 3. $f(\sigma) \simeq f\cdot\sigma$.

The morphism induced by an orchestrator f is monotone with respect to the strong subcontract relation and is well behaved with respect to the choice operators.

Theorem 4. *The following properties hold: (1) $\sigma \sqsubseteq \tau$ implies $f(\sigma) \sqsubseteq f(\tau)$; (2) $f(\sigma)+f(\tau) \sqsubseteq f(\sigma+\tau)$; (3) $f(\sigma)\oplus f(\tau) \simeq f(\sigma\oplus\tau)$.*

Observe that $f(\sigma)+f(\tau) \simeq f(\sigma+\tau)$ does not hold in general, because of the asynchronous actions that f may offer to the client side. Consider for example $f \stackrel{\text{def}}{=} \langle a,\varepsilon\rangle.(\langle\overline{b},b\rangle + \langle\overline{d},d\rangle)$. Then $f(\overline{b})+f(\overline{d}) = a.\overline{b}+a.\overline{d} \simeq a.(\overline{b}\oplus\overline{d}) \sqsubseteq a.(\overline{b}+\overline{d}) = f(\overline{b}+\overline{d})$ but $f(\overline{b}+\overline{d}) \not\sqsubseteq f(\overline{b})+f(\overline{d})$. Nonetheless Theorem 4 allows us to prove an interesting property of $\preceq$: if $\sigma \sqsubseteq f\cdot\sigma'$ and $\tau \sqsubseteq f\cdot\tau'$, then $\sigma+\tau \sqsubseteq f\cdot(\sigma'+\tau')$ and $\sigma\oplus\tau \sqsubseteq f\cdot(\sigma'\oplus\tau')$. This means that if $\sigma \preceq \sigma'$ and $\tau \preceq \tau'$ and the two relations are witnessed by the *same* orchestrator, then $\sigma+\tau \preceq \sigma'+\tau'$ and $\sigma\oplus\tau \preceq \sigma'\oplus\tau'$. In other words, a sufficient condition for being able to orchestrate $\sigma'+\tau'$ is that the orchestrator must be independent of the branch (either σ' or τ') taken by the service, which is in fact the minimum requirement we could expect. In general however $\preceq$ is not a precongruence: $a \preceq a+b.c$ but $a+b.d \not\preceq a+b.c+b.d \simeq a+b.(c\oplus d)$.

Composition of orchestrators. Transitivity of $\preceq$ is not granted by its definition, because $\sigma \preceq \tau$ means that every client that is *strongly* compliant with σ is also *weakly* compliant with τ. So it is not clear whether $\sigma \preceq \tau$ and $\tau \preceq \sigma'$ implies $\sigma \preceq \sigma'$. By Theorem 2, we know that there exist f and g such that $\sigma \sqsubseteq f\cdot\tau$ and $\tau \sqsubseteq g\cdot\sigma'$. Furthermore, by Theorem 4(1) and transitivity of $\sqsubseteq$ we deduce that $\sigma \sqsubseteq f\cdot\tau \sqsubseteq f\cdot g\cdot\sigma'$. Thus we can conclude $\sigma \preceq \sigma'$ provided that for any two orchestrators f and g it is possible to find an orchestrator $f\cdot g$ such that $f\cdot g\cdot\sigma' \simeq (f\cdot g)\cdot\sigma'$. Alternatively, by considering orchestrators f and g as morphisms, we are asking whether their functional composition $f\circ g$ is still an orchestrator. This is not the case in general. To see why, consider $f \stackrel{\text{def}}{=} \langle a,\varepsilon\rangle.\langle c,\varepsilon\rangle.(\langle\varepsilon,\overline{a}\rangle.\langle\overline{b},b\rangle + \langle\varepsilon,\overline{c}\rangle.\langle\overline{d},d\rangle)$ and $g \stackrel{\text{def}}{=} \langle a,\varepsilon\rangle.\langle\overline{b},b\rangle + \langle c,\varepsilon\rangle.\langle\overline{d},d\rangle$ and apply them to the contract $\sigma \stackrel{\text{def}}{=} \overline{b}+\overline{d}$. We have $f\cdot g\cdot\sigma \simeq f(g(\sigma)) \simeq f(a.\overline{b}+c.\overline{d}) \simeq a.c.(\overline{b}\oplus\overline{d})$. The subsequent applications of g first and then f introduce some nondeterminism due to the uncertainty as to which synchronization (on a or on c) will happen. This uncertainty yields the internal choice $\overline{b}\oplus\overline{d}$ in the resulting contract. No single orchestrator can turn $\overline{b}+\overline{d}$ into $a.c.(\overline{b}\oplus\overline{d})$ for orchestrators do not manifest internal nondeterminism. The problem could be addressed by adding internal nondeterminism to the orchestration language, but this seems quite artificial and, as a matter of facts, is unnecessary. If we just require that $f\cdot g\cdot\sigma \sqsubseteq (f\cdot g)\cdot\sigma$, then $\sigma \sqsubseteq (f\cdot g)\cdot\sigma'$ follows by transitivity of $\sqsubseteq$. The orchestrator $f\cdot g$ can be defined as

$$f\cdot g \stackrel{\text{def}}{=} \sum_{f \xrightarrow{\langle\alpha,\varepsilon\rangle} f'} \langle\alpha,\varepsilon\rangle.(f'\cdot g) + \sum_{g \xrightarrow{\langle\varepsilon,\overline{\alpha}\rangle} g'} \langle\varepsilon,\overline{\alpha}\rangle.(f\cdot g') + \sum_{f \xrightarrow{\langle\varphi,\overline{\alpha}\rangle} f',\, g \xrightarrow{\langle\alpha,\varphi'\rangle} g',\, \varphi\varphi'\neq\varepsilon} \langle\varphi,\varphi'\rangle.(f'\cdot g') + \sum_{f \xrightarrow{\langle\varepsilon,\overline{\alpha}\rangle} f',\, g \xrightarrow{\langle\alpha,\varepsilon\rangle} g'} (f'\cdot g')$$

The first two subterms in the definition of $f \cdot g$ indicate that all the asynchronous actions offered by f (respectively, g) to the client (respectively, service) are available. The third subterm turns synchronous actions into asynchronous ones: for example, $\langle \alpha, \overline{\alpha} \rangle \cdot \langle \alpha, \varepsilon \rangle = \langle \alpha, \varepsilon \rangle$ and $\langle \varepsilon, \overline{\alpha} \rangle \cdot \langle \alpha, \overline{\alpha} \rangle = \langle \varepsilon, \alpha \rangle$. The last subterm accounts for the "synchronizations" occurring within the orchestrator, when f and g exchange a message and the two actions annihilate each other. If we consider the orchestrators f and g defined above, we obtain $f \cdot g = \langle a, \varepsilon \rangle . \langle c, \varepsilon \rangle . (\langle \overline{b}, b \rangle + \langle \overline{d}, d \rangle)$ and we observe $(f \cdot g)(\overline{b} + \overline{d}) = a.c.(\overline{b} + \overline{d})$. The following result proves that $f \cdot g$ is correct and, as a corollary, that $\preceq$ is a preorder:

Theorem 5. $f \cdot g \cdot \sigma \sqsubseteq (f \cdot g) \cdot \sigma$.

Against the objection that $f \cdot g$ is "more powerful" than $f \circ g$ it is sufficient to observe that if f and g are k-orchestrators, then $f \cdot g$ is a $2k$-orchestrator. Thus, $f \cdot g$ is nothing more than some proper combination of f and g, as expected.

4 Contract Duality with Orchestration

We tackle the problem of finding the dual contract $\rho^{\perp}$ of a given client contract ρ. Recall that $\rho^{\perp}$ should be the smallest (according to $\preceq$) contract such that ρ is compliant with $\rho^{\perp}$. Without loss of generality, we restrict the definition of the dual contract to so-called *canonical client contracts*. A client contract is canonical if every derivation leading to $\mathtt{0}$ emits $\mathtt{e}$ as its last visible action. Formally, ρ is canonical if $\rho \stackrel{\varphi}{\Longrightarrow} \rho'$ and $\rho' \simeq \mathtt{0}$ implies $\varphi = \varphi' \mathtt{e}$ for some φ'. This way we avoid client contracts such as $a + b.\mathtt{e}$ which can fail if synchronizing on a. The subterm a indicates that the client is unable of handling a, thus the occurrence of a in the contract is useless as far as querying is concerned and it can be safely ignored.

Definition 8 (dual contract). *Let ρ be a canonical client contract. The* dual contract *of ρ, denoted by $\rho^{\perp}$, is defined as:*

$$\rho^{\perp} \stackrel{\text{def}}{=} \sum_{\rho \Downarrow \mathtt{R}, \mathtt{e} \notin \mathtt{R}} \bigoplus_{\alpha \in \mathtt{R}} \overline{\alpha}.\rho(\alpha)^{\perp}$$

The idea of the dual operator is to consider every state $\mathtt{R}$ of the client in which the client cannot terminate successfully ($\mathtt{e} \notin \mathtt{R}$). For every such state the service must provide at least one way for the client to proceed, and the least service that guarantees this is given by the internal choice of all the co-actions in $\overline{\mathtt{R}}$ (note that $\mathtt{R} \neq \emptyset$ because the client is canonical). For example $(a.\mathtt{e})^{\perp} = (a.\mathtt{e} \oplus \mathtt{e})^{\perp} = \overline{a}$ (the service must provide $\overline{a}$); $(a.\mathtt{e} + \mathtt{e})^{\perp} = \mathtt{0}$ (the service need not provide anything because the client can terminate immediately); $(a.\mathtt{e} + b.\mathtt{e})^{\perp} = \overline{a} \oplus \overline{b}$ (the service can decide whether to provide $\overline{a}$ or $\overline{b}$); $(\mathtt{rec}\ x.a.x)^{\perp} \simeq \mathtt{rec}\ x.\overline{a}.x$ (the service must provide an infinite sequence of $\overline{a}$'s).

Theorem 6 (duality). *Let ρ be a canonical client contract. Then (1) $\rho \dashv \rho^{\perp}$ and (2) $\rho \dashv \sigma$ implies $\rho^{\perp} \preceq \sigma$.*

The assumption of using orchestrators is essential as far as duality is concerned: $(a.\mathtt{e} + \mathtt{e})^{\perp} = \mathtt{0}$ but $\mathtt{0}$ is *not* the smallest (according to $\sqsubseteq$) contract satisfying $a.\mathtt{e} + \mathtt{e}$. For example, $\mathtt{0} \oplus b \sqsubseteq \mathtt{0}$ and $a.\mathtt{e} + \mathtt{e} \dashv \mathtt{0} \oplus b$. On the contrary, $\mathtt{0}$ is the least element of $\preceq$ and it can be used in place of any service contract that exposes an empty ready set.

5 Synthesizing Orchestrators

In this section we devise an algorithm for computing the k-orchestrator witnessing $\sigma \preceq \tau$, provided there is one. The algorithm is defined inductively by the following rules:

$$\text{(A1)}\quad \frac{\begin{array}{c}\mathrm{A} \subseteq \{\langle \varphi, \overline{\varphi}' \rangle \mid \sigma \stackrel{\varphi}{\Longrightarrow}, \tau \stackrel{\varphi'}{\Longrightarrow}, \mathbb{B} \vdash_k \langle \varphi, \overline{\varphi}' \rangle\} \qquad x \text{ fresh} \\ \tau \Downarrow \mathrm{S} \Rightarrow (\exists \mathrm{R} : \sigma \Downarrow \mathrm{R} \wedge \mathrm{R} \subseteq \mathrm{A} \circ \mathrm{S}) \vee (\emptyset \bullet \mathrm{A}) \cap \overline{\mathrm{S}} \neq \emptyset \\ \Gamma \cup \{(\mathbb{B}, \sigma, \tau) \mapsto x\}, \mathbb{B} \langle \varphi, \overline{\varphi}' \rangle \vdash_k f_{\langle \varphi, \overline{\varphi}' \rangle} : \sigma(\varphi) \preceq^{\mathtt{a}} \tau(\varphi') \ {}^{\forall \langle \varphi, \overline{\varphi}' \rangle \in \mathrm{A}}\end{array}}{\Gamma, \mathbb{B} \vdash_k \mathtt{rec}\, x. \textstyle\sum_{\mu \in \mathrm{A}} \mu. f_\mu : \sigma \preceq^{\mathtt{a}} \tau} \qquad \text{(A2)}\quad \frac{\Gamma(\mathbb{B}, \sigma, \tau) = x}{\Gamma, \mathbb{B} \vdash_k x : \sigma \preceq^{\mathtt{a}} \tau}$$

A judgment of the form $\Gamma, \mathbb{B} \vdash_k f : \sigma \preceq^{\mathtt{a}} \tau$ means that f is a k-orchestrator proving that $\sigma \preceq \tau$ when the buffer of the orchestrator is in state $\mathbb{B}$. The context Γ memoizes triples $(\mathbb{B}, \sigma, \tau)$ so as to guarantee termination (see Proposition 1 below). The k-buffer $\mathbb{B}$ keeps track of the past history of the orchestrator (which messages the orchestrator has accepted and not yet delivered). We write $f : \sigma \preceq^{\mathtt{a}}_k \tau$ if $\emptyset, \tilde{\emptyset} \vdash_k f : \sigma \preceq^{\mathtt{a}} \tau$.

Although rule (A1) looks formidable, it is a straightforward adaptation of the conditions in Definition 4. Recall that the purpose of the algorithm is to find whether there exists an orchestrator f such that every client strongly compliant with σ is weakly compliant with τ when this service is orchestrated by f. Since $\mathbb{B}$ is a k-buffer, there is a finite number of useful asynchronous orchestration actions that can be enabled: an action $\langle \overline{a}, \varepsilon \rangle$ is enabled only if $\mathbb{B}(\circ, \overline{a}) > 0$; an action $\langle a, \varepsilon \rangle$ is enabled only if the buffer has not reached its capacity, namely if $\mathbb{B}(\bullet, \overline{a}) < k$; symmetrically for asynchronous service actions. Also, it is pointless to consider any orchestration action that would not cause any synchronization to occur. Hence, the set $\{\langle \varphi, \overline{\varphi}' \rangle \mid \sigma \stackrel{\varphi}{\Longrightarrow}, \tau \stackrel{\varphi'}{\Longrightarrow}, \mathbb{B} \vdash_k \langle \varphi, \overline{\varphi}' \rangle\}$ of useful, enabled orchestration actions in the first premise of the rule is finite. Of all of such actions, the algorithm considers only those in some subset A such that the execution of any orchestration action in A does not lead to a deadlock later on during the interaction. This is guaranteed if for every $\langle \varphi, \overline{\varphi}' \rangle \in \mathrm{A}$ we are able to find an orchestrator f_μ that proves $\tau(\varphi') \preceq \sigma(\varphi)$ (fourth premise of the rule). When checking the continuations, the context Γ is augmented associating the triple $(\mathbb{B}, \sigma, \tau)$ with a fresh orchestrator variable x, and the buffer is updated to account for the orchestration action just occurred. If the set A is large enough so that τ can be made to look like a more deterministic version of σ (third premise of the rule), then σ and τ can be related. The orchestrator computed in the conclusion of rule (A1) offers the union of all the useful, enabled orchestration actions μ, each one followed by the corresponding f_μ continuation. Rule (A2) is used when the algorithm needs to check whether there exists f such that $\Gamma, \mathbb{B} \vdash_k f : \sigma \preceq^{\mathtt{a}} \tau$ and $(\mathbb{B}, \sigma, \tau) \in \mathtt{dom}(\Gamma)$. In this case $\Gamma(\mathbb{B}, \sigma, \tau)$ is a variable that represents the orchestrator that the algorithm has already determined for proving $\sigma \preceq \tau$.

Theorem 7. *The following properties hold:*

1. *(termination) it is decidable to check whether there exists f such that $f : \sigma \preceq^{\mathtt{a}}_k \tau$;*
2. *(correctness) $f : \sigma \preceq^{\mathtt{a}}_k \tau$ implies that f has rank k and $\sigma \sqsubseteq f \cdot \tau$;*
3. *(completeness) $\sigma \preceq \tau$ implies $f : \sigma \preceq^{\mathtt{a}}_k \tau$ for some k and some f of rank k.*

6 An Example: Orchestrated Dining Philosophers

Consider a variant of the problem of the dining philosophers in which a service provider hires two philosophers for providing philosophical thoughts to the clients of the service. Each philosopher is modeled by the following contract:

$$P_i \stackrel{\text{def}}{=} \texttt{rec}\ x.fork_i.fork_i.\overline{thought}.\overline{fork}.\overline{fork}.x$$

where the $fork_i$ actions model the philosopher's request of two forks, $\overline{thought}$ models the production of a thought, and the $\overline{fork}$ actions notify the client that the forks are returned. We decorate $fork_i$ actions with an index i for distinguishing fork requests coming from different philosophers. Also, we need some way for describing the contract of two philosophers running in parallel. To this aim we make use of a parallel composition operator over contracts so that $\sigma \mid \tau$ stands for the interleaving of all the actions in σ and τ. Assuming that σ and τ never synchronize with each other, the $\mid$ operator can be expressed using a simplified form of *expansion law* [16]:

$$\sigma \mid \tau \stackrel{\text{def}}{=} \bigoplus_{\sigma\Downarrow \mathrm{R}, \tau\Downarrow \mathrm{S}} (\sum_{\alpha\in \mathrm{R}} \alpha.(\sigma(\alpha) \mid \tau) + \sum_{\alpha\in \mathrm{S}} \alpha.(\sigma \mid \tau(\alpha)))$$

The client modeled by the contract

$$C \stackrel{\text{def}}{=} \texttt{rec}\ x.\sum_{i=1..2}\overline{fork_i}.\sum_{i=1..2}\overline{fork_i}.thought.fork.fork.x$$

expects to be able to receive thoughts forever, without ever getting stuck. The problem of this sloppy client is that it does not care that the two forks it provides end up to the same philosopher and this may cause the system to deadlock. To see whether such client can be made compliant with $P_1 \mid P_2$ we compute its dual contract

$$C^{\perp} \simeq \texttt{rec}\ x.\bigoplus_{i=1..2} fork_i.\bigoplus_{i=1..2} fork_i.\overline{thought}.\overline{fork}.\overline{fork}.x$$

and then we check whether $C^{\perp} \preceq P_1 \mid P_2$ using the algorithm. If we consider the sequence of actions $\overline{fork}_1\overline{fork}_2$ we reduce to checking whether $\overline{thought}.\overline{fork}.\overline{fork}.C^{\perp} \preceq P_1(fork_1) \mid P_2(fork_2)$. Now $P_1(fork_1) \mid P_2(fork_2)$ has just the ready set $\{fork_1, fork_2\}$, while the residual of the client's dual contract has just the ready set $\{\overline{thought}\}$. There is no orchestration action that can let the algorithm make some progress from this state. Thus the algorithm finds out that the two forks sent by the client must be delivered to the same philosopher, and this is testified by the resulting orchestrator

$$f \stackrel{\text{def}}{=} \texttt{rec}\ x.\sum_{i=1..2}\langle fork_i,\overline{fork_i}\rangle.\langle fork_i,\overline{fork_i}\rangle.\langle thought,\overline{thought}\rangle.\langle fork,\overline{fork}\rangle.\langle fork,\overline{fork}\rangle.x$$

Suppose now that the service provider is forced to update the service with two new philosophers who, according to their habit, produce their thoughts only after having returned the forks. Their behavior can be described by the contract

$$Q_i \stackrel{\text{def}}{=} \texttt{rec}\ x.fork_i.fork_i.\overline{fork}.\overline{fork}.\overline{thought}.x$$

The service provider may wonder whether the clients of the old service will still be satisfied by the new one. The problem can be formulated as checking whether $P_1 \mid P_2 \preceq$

$Q_1 \mid Q_2$ and the interesting step is when the algorithm eventually checks $P_1(\mathit{fork}_1\mathit{fork}_1) \mid P_2 \preceq Q_1(\mathit{fork}_1\mathit{fork}_1) \mid Q_2$ (symmetrically for P_2 and the sequence of actions $\mathit{fork}_2\mathit{fork}_2$). At this stage $P_1(\mathit{fork}_1\mathit{fork}_1) \mid P_2$ has just the ready set $\{\overline{\mathit{thought}}, \mathit{fork}_2\}$, whereas the contract $Q_1(\mathit{fork}_1\mathit{fork}_1) \mid Q_2$ has just the ready set $\{\overline{\mathit{fork}}, \mathit{fork}_2\}$. By accepting the two $\overline{\mathit{fork}}$ messages asynchronously we reduce to checking whether $P_1(\mathit{fork}_1\mathit{fork}_1) \mid P_2 \preceq \overline{\mathit{thought}}.Q_1 \mid Q_2$, which holds by allowing the $\overline{\mathit{thought}}$ action to occur, followed by the asynchronous sending of the two buffered $\overline{\mathit{fork}}$ messages. Overall the relation is proved by the orchestrator

$$g \stackrel{\text{def}}{=} \mathtt{rec}\, x.\, \textstyle\sum_{i=1..2} \langle \mathit{fork}_i, \overline{\mathit{fork}_i} \rangle . \sum_{i=1..2} \langle \mathit{fork}_i, \overline{\mathit{fork}_i} \rangle . \langle \varepsilon, \mathit{fork} \rangle . \langle \varepsilon, \mathit{fork} \rangle . \\ \langle \mathit{thought}, \overline{\mathit{thought}} \rangle . \langle \overline{\mathit{fork}}, \varepsilon \rangle . \langle \overline{\mathit{fork}}, \varepsilon \rangle . x$$

and now the sloppy C client will be satisfied by the service $(f \cdot g) \cdot (Q_1 \mid Q_2)$.

7 Discussion

We have adapted the testing framework [11, 16] by assuming that orchestrators can mediate the interaction between a client and a service. We have been able to define a decidable, liveness-preserving subcontract relation that is coarser than the existing ones, thus enlarging the set of services satisfying a given client and favoring service reuse. Unlike other orchestration languages, the features of simple orchestrators language stem directly from the equivalence relation, rather than being designed *a priori*.

The synthesis algorithm as it stands is computationally expensive. It is well known that deciding $\sqsubseteq$ is PSPACE-complete [20] although common practice suggests that worst cases occur seldom [9]. In our setting more complexity is added for synthesizing the orchestrator and the algorithm requires one to guess the rank of the orchestrator needed for relating two contracts σ and τ. We plan to study a variant of the algorithm that is able to discover the best (of smallest rank) orchestrator that proves $f : \sigma \preceq^{\mathsf{a}}_{k} \tau$ (an upper bound can be established by exploiting contract regularity) in the spirit of what has been done in [6] with so-called "best" filters.

The observations of §3, where we present orchestrators as morphisms for relating otherwise incompatible behavioral types, deserve further investigation. In particular, we plan to study a class of *invertible orchestrators* characterizing isomorphic contracts, much like *invertible functions* are used in [13, 24] for characterizing isomorphic types in a functional language.

Asynchronous variants of the classical testing preorders [4, 7] are notoriously more involved than their synchronous counterparts and they are usually defined assuming that self-synchronization is possible and that output messages are allowed to float around in unbounded buffers. Since these assumptions do not reflect the practice of Web services, our development can be seen as a practical variant of the classical asynchronous testing theories. In particular, it might be possible to reduce the asynchronous *must* preorder without self-synchronization to our subcontract relation by analyzing the structure of orchestrators proving the relation (an orchestrator that always enables all of its asynchronous input and output actions acts like an unbounded buffer).

References

1. Alves, A., Arkin, A., Askary, S., Barreto, C., et al.: Web Services Business Process Execution Language Version 2.0 (2007)
2. Banerji, A., Bartolini, C., Beringer, D., Chopella, V., et al.: Web Services Conversation Language (WSCL) 1.0 (2002)
3. Beringer, D., Kuno, H., Lemon, M.: Using WSCL in a UDDI Registry 1.0 (2001)
4. Boreale, M., Nicola, R.D., Pugliese, R.: Trace and testing equivalence on asynchronous processes. Information and Computation 172(2), 139–164 (2002)
5. 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)
6. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for Web services. In: Proceedings of POPL 2008, pp. 261–272. ACM, New York (2008)
7. Castellani, I., Hennessy, M.: Testing theories for asynchronous languages. In: Arvind, V., Ramanujam, R. (eds.) FST TCS 1998. LNCS, vol. 1530, pp. 90–101. Springer, Heidelberg (1998)
8. Chinnici, R., Moreau, J.-J., Ryman, A., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language (2007)
9. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. Formal Aspects of Computing 5(1), 1–20 (1993)
10. Colgrave, J., Januszewski, K.: Using WSDL in a UDDI registry, version 2.0.2. Technical note, OASIS (2004)
11. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34, 83–133 (1984)
12. 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)
13. Di Cosmo, R.: Isomorphisms of Types: from Lambda Calculus to Information Retrieval and Language Design. Birkhäuser, Basel (1995)
14. Fournet, C., Hoare, C.A.R., Rajamani, S.K., Rehof, J.: Stuck-free conformance. Technical Report MSR-TR-2004-69, Microsoft Research (2004)
15. Gay, S., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica 42(2-3), 191–225 (2005)
16. Hennessy, M.: Algebraic Theory of Processes. In: Foundation of Computing. MIT Press, Cambridge (1988)
17. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
18. Inverardi, P., Tivoli, M.: Software architecture for correct components assembly. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 92–121. Springer, Heidelberg (2003)
19. Inverardi, P., Tivoli, M.: A reuse-based approach to the correct and automatic composition of web-services. In: ESSPE 2007, pp. 29–33 (2007)
20. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation 86(1), 43–68 (1990)
21. 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)

22. Misra, J., Cook, W.R.: Computation orchestration – a basis for wide-area computing. Software and Systems Modeling 6(1), 83–110 (2007)
23. Padovani, L.: Contract-directed synthesis of simple orchestrators. Technical report (2008), http://www.sti.uniurb.it/padovani/Papers/OrchestratorSynthesis.pdf
24. Rittri, M.: Retrieving library functions by unifying types modulo linear isomorphism. RAIRO Theoretical Informatics and Applications 27(6), 523–540 (1993)
25. von Riegen, C., Trickovic, I.: Using bpel4ws in a UDDI registry. Technical note, OASIS (2004)

Environment Assumptions for Synthesis

Krishnendu Chatterjee[2], Thomas A. Henzinger[1], and Barbara Jobstmann[1]

[1] EPFL, Lausanne
[2] University of California, Santa Cruz

Abstract. The synthesis problem asks to construct a reactive finite-state system from an ω-regular specification. Initial specifications are often unrealizable, which means that there is no system that implements the specification. A common reason for unrealizability is that assumptions on the environment of the system are incomplete. We study the problem of correcting an unrealizable specification φ by computing an environment assumption ψ such that the new specification $\psi \rightarrow \varphi$ is realizable. Our aim is to construct an assumption ψ that constrains only the environment and is as weak as possible. We present a two-step algorithm for computing assumptions. The algorithm operates on the game graph that is used to answer the realizability question. First, we compute a safety assumption that removes a minimal set of environment edges from the graph. Second, we compute a liveness assumption that puts fairness conditions on some of the remaining environment edges. We show that the problem of finding a minimal set of fair edges is computationally hard, and we use probabilistic games to compute a locally minimal fairness assumption.

1 Introduction

Model checking has become a successful verification technique in hardware and software design. Although the method is automated, the success of a verification process highly depends on the quality of the specification. Writing correct and complete specifications is a tedious task: it usually requires several iterations until a satisfactory specification is obtained. Specifications are often too weak (e.g., they may be vacuously satisfied [2,14]); or too strong (e.g., they may allow too many environment behaviors), resulting in spurious counterexamples. In this work we automatically strengthen the environment constraints within specifications whose assumptions about the environment behavior are so weak as to make it impossible for a system to satisfy the specification.

Automatically deriving environment assumptions has been studied from several points of view. For instance, in circuit design one is interested in automatically constructing environment models that can be used in test-bench generation [21,19]. In compositional verification, environment assumptions have been generated as the weakest input conditions under which a given software or hardware component satisfies a given specification [4,6]. We follow a different path by leaving the design out of the picture and deriving environment assumptions from the specification alone. Given a specification, we aim to compute a least restrictive environment that allows for an implementation of the specification. The assumptions that we compute can assist the designer in different ways. They can be used as baseline necessary conditions in component-based model checking. They can be used in designing interfaces and generating test cases

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 147–161, 2008.

for components before the components themselves are implemented. They can provide insights into the given specification. And above all, in the process of automatically constructing an implementation for the given specification ("synthesis"), they can be used to correct the specification in a way that makes implementation possible.

While specifications of closed systems can be implemented if they are *satisfiable*, specifications of open systems can be implemented if they are *realizable* —i.e., there is a system that satisfies the specification without constraining the inputs. The key idea of our approach is that given a specification φ, if φ is not realizable, it cannot be complete and has to be weakened by introducing assumptions on the environment of the system. We do this by computing an assumption ψ such that the new specification $\psi \rightarrow \varphi$ is realizable. Our aim is to construct a condition ψ that does not constrain the system and is as weak as possible. The notion that ψ must constrain only the environment can be captured by requiring that ψ itself is realizable for the environment —i.e., there exists an environment that satisfies ψ without constraining the outputs of the system. The notion that ψ be as weak as possible is more difficult to capture. We will show that in certain situations, there is no unique weakest environment-realizable assumption ψ, and in other situations, it is NP-hard to compute such an assumption.

Example. During a typical effort of formally specifying hardware designs [5], some specifications were initially not realizable. One specification that was particularly difficult to analyze can be simplified to the following example. Consider a system with two input signals $\mathtt{r}$ and $\mathtt{c}$, and one output signal $\mathtt{g}$. The specification requires that (i) every request is eventually granted starting from the next time step, written in linear temporal logic as $\square(\mathtt{r} \rightarrow \bigcirc\lozenge\mathtt{g})$; and (ii) whenever $\mathtt{c}$ or $\mathtt{g}$ are high, then $\mathtt{g}$ has to stay low in the next time step, written $\square((\mathtt{c} \vee \mathtt{g}) \rightarrow \bigcirc\neg\mathtt{g})$. This specification is not realizable because the environment can force, by sending $\mathtt{c}$ all the time, that $\mathtt{g}$ has to stay low forever (Part (ii)). Thus requests cannot be answered, and Part (i) is violated.

One assumption that makes this specification realizable is $\psi_1 = \square\neg\mathtt{c}$. This assumption is undesirable because it forbids the environment to send $\mathtt{c}$. A system synthesized with this assumption would ignore the signal $\mathtt{c}$. Assumptions $\psi_2 = \square\lozenge\neg\mathtt{c}$ and $\psi_3 = \square(\mathtt{r} \rightarrow \lozenge\neg\mathtt{c})$ are more desirable but still not satisfactory: ψ_2 forces the environment to lower $\mathtt{c}$ infinitely often even when no requests are sent, and ψ_3 is not strong enough to implement a system that in each step first produces an output and then reads the input. Assume that the system starts with output $\mathtt{g} = 0$ in time step 0, then receives the input $\mathtt{r} = 1$ and $\mathtt{c} = 0$, now in time step 1, it can choose between (a) $\mathtt{g} = 1$, or (b) $\mathtt{g} = 0$. If it chooses to set grant to high by (a), then the environment can provide the same inputs once more ($\mathtt{r} = 1$ and $\mathtt{c} = 0$) and can set all subsequent inputs to $\mathtt{r} = 0$ and $\mathtt{c} = 1$. Then the environment has satisfied ψ_3 because during the two requests in time step 0 and 1 the signal $\mathtt{c}$ was kept low, but the system cannot fulfill Part (i) of its specification without violating Part (ii) due to $\mathtt{g} = 1$ in time step 1 and $\mathtt{c} = 1$ afterwards. On the other hand, if the system decides to choose to set $\mathtt{g} = 0$ by (b), then the environment can choose to set the inputs to $\mathtt{r} = 0$ and $\mathtt{c} = 1$ and the system again fails to fulfill Part (i) without violating (ii). The assumption $\psi_4 = \square(\mathtt{r} \rightarrow \bigcirc\lozenge\neg\mathtt{c})$, which is a subset of ψ_3, is sufficient. However, there are infinitely many sufficient assumptions between ψ_3 and ψ_4, such as $\psi_3' = (\neg\mathtt{c} \wedge \bigcirc\psi_3) \vee \psi_3$. The assumption $\psi_5 = \square(\mathtt{r} \rightarrow \bigcirc\lozenge(\neg\mathtt{c} \vee \mathtt{g}))$ is also weaker than ψ_3 and still sufficient, because the environment only needs to lower

c eventually if a request has not been answered yet. Finally, let $\xi = \mathtt{r} \rightarrow \bigcirc\Diamond(\neg\mathtt{c} \vee \mathtt{g})$ and consider the assumption $\psi_6 = \xi\,\mathsf{W}(\xi \wedge (\mathtt{c} \vee \mathtt{g}) \wedge \bigcirc\mathtt{g})$, which is a sufficient assumption (where W is the *weak-until* operator of LTL). It is desirable because it states that whenever a request is sent, the environment has to eventually lower c if it has not seen the signal g, but as soon as the system violates its specification (Part (ii)) all restrictions on the environment are dropped. If we replace ξ in ψ_6 with $\xi' = \mathtt{r} \rightarrow \Diamond(\neg\mathtt{c} \vee \mathtt{g})$, we get again an assumption that is not sufficient for the specification to be realizable. This example shows that the notion of weakest and desirable are hard to capture.

Contributions. The realizability problem (and synthesis problem) can be reduced to emptiness checking for tree automata, or equivalently, to solving turn-based two-player games on graphs [17]. More specifically, an ω-regular specification φ is realizable iff there exists a winning strategy in a certain parity game constructed from φ. If φ is not realizable, then we construct an environment assumption ψ such that $\psi \rightarrow \varphi$ is realizable, in two steps. First, we compute a safety assumption that removes a minimal set of environment edges from the game graph. Second, we compute a liveness assumption that puts fairness conditions on some of the remaining environment edges of the game graph: if these edges can be chosen by the environment infinitely often, then they need to be chosen infinitely often. While the problem of finding a minimal set of fair edges is shown to be NP-hard, a local minimum can be found in polynomial time (in the size of the game graph) for Büchi specifications, and in NP $\cap$ coNP for parity specifications. The algorithm for checking the sufficiency of a set of fair edges is of independent theoretical interest, as it involves a novel reduction of deterministic parity games to probabilistic parity games. We show that the resulting conjunction of safety and liveness assumptions is sufficient to make the specification realizable, and itself realizable by the environment. We also illustrate the algorithm on several examples, showing that it computes natural assumptions.

Related work. There are some related works that consider games that are not winning, methods of restricting the environment, and constructing most general winning strategies in games. The work of [11] considers games that are not winning, and considers *best-effort* strategies in such games. However, relaxing the winning objective to make the game winning is not considered. In [8], a notion of nonzero-sum game is proposed, where the strategies of the environment are restricted according to a given objective, but the paper does not study how to obtain an environment objective that is sufficient to transform the game to a winning one. A minimal assumption on a player with an objective can be captured by the most general winning strategy for the objective. The results of [3] show that such most general winning strategies exist only for safety games, and also present an approach to compute a strategy, called a *permissive strategy*, which subsumes behavior of all memoryless winning strategies. Our approach is different, as we attempt to construct the minimal environment assumption that makes a game winning.

Outline. In Section 2, we introduce the necessary theoretical background for defining and computing environment assumptions. Section 3 discusses environment assumptions and why they are difficult to capture. In Sections 4 and 5, we compute, respectively, safety and liveness assumptions, which are then combined in Section 6. A full version with detailed proofs can be found in [7].

2 Preliminaries

Words, languages, safety, and liveness. Given a finite alphabet Σ and an infinite word $w \in \Sigma^\omega$, we use w_i to denote the $(i+1)^{th}$ letter of w, and w^i to denote the finite prefix of w of length $i+1$. Given a word $w \in \Sigma^\omega$, we write $\mathrm{odd}(w)$ for the subsequence of w consisting of the odd positions ($\forall i \geq 0 : \mathrm{odd}(w)_i = w_{2i+1}$). Given a set $L \subseteq \Sigma^\omega$ of infinite words, we define the set of finite prefixes by $\mathrm{pref}(L) = \{v \in \Sigma^* \mid \exists w \in L, i \geq 0 : v = w^i\}$. Given a set $L \subseteq \Sigma^*$ of finite words, we define the set of infinite limits by $\mathrm{safe}(L) = \{w \in \Sigma^\omega \mid \forall i \geq 0 : w^i \in L\}$. A language $L \subseteq \Sigma^\omega$ is a *safety* language if $L = \mathrm{safe}(\mathrm{pref}(L))$. A language $L \subseteq \Sigma^\omega$ is a *liveness* language if $\mathrm{pref}(L) = \Sigma^*$. Every ω-regular language $L \subseteq \Sigma^\omega$ can be presented as the intersection of the safety language $L_S = \mathrm{safe}(\mathrm{pref}(L))$ and the liveness language $L_L = (\Sigma^\omega \setminus L_S) \cup L$ [1].

Transducers. We model reactive systems as deterministic finite-state transducers. We fix a finite set P of atomic propositions, and a partition of P into a set O of output and a set I of input propositions. We use the alphabets $\Sigma = 2^P$, $\mathcal{O} = 2^O$, and $\mathcal{I} = 2^I$. A *Moore transducer* with input alphabet $\mathcal{I}$ and output alphabet $\mathcal{O}$ is a tuple $\mathcal{T} = (Q, q_I, \Delta, \kappa)$, where Q is a finite set of states, $q_I \in Q$ is the initial state, Δ: $Q \times \mathcal{I} \to Q$ is the transition function, and κ: $Q \to \mathcal{O}$ is a state labeling function. A *Mealy transducer* is like a Moore transducer, except that κ: $Q \times \mathcal{I} \to \mathcal{O}$ is a transition labeling function. A Moore transducer describes a reactive system that reads words over the alphabet $\mathcal{I}$ and writes words over the alphabet $\mathcal{O}$. The environment of the system, in turn, can be described by a Mealy transducer with input alphabet $\mathcal{O}$ and output alphabet $\mathcal{I}$. We extend the transition function Δ to finite words $w \in \mathcal{I}^*$ inductively by $\Delta(q, w) = \Delta(\Delta(q, w^{|w|-1}), w_{|w|})$ for $|w| > 0$. Given a word $w \in \mathcal{I}^\omega$, the run of $\mathcal{T}$ over w is the infinite sequence $\pi \in Q^\omega$ of states such that $\pi_0 = q_I$, and $\pi_{i+1} = \Delta(\pi_i, w_i)$ for all $i \geq 0$. The run π over w generates the infinite word $\mathcal{T}(w) \in \Sigma^\omega$ defined by $\mathcal{T}(w)_i = \kappa(\pi_i) \cup w_i$ for all $i \geq 0$ in the case of Moore transducers; and $\mathcal{T}(w)_i = \kappa(\pi_i, w_i) \cup w_i$ for all $i \geq 0$ in the case of Mealy transducers. The *language* of $\mathcal{T}$ is the set $L(\mathcal{T}) = \{\mathcal{T}(w) \mid w \in \mathcal{I}^\omega\}$ of all generated infinite words.

Specifications and realizability. A *specification* of a reactive system is an ω-regular language $L \subseteq \Sigma^\omega$. We use Linear Temporal Logic (LTL) formulae over the atomic proposition P, as well as ω-automata with transition labels from Σ, to define specifications. Given an LTL formula (resp. ω-automaton) φ, we write $L(\varphi) \subseteq \Sigma^\omega$ for the set of infinite words that satisfy (resp. are accepted by) φ. A transducer $\mathcal{T}$ *satisfies* a specification $L(\varphi)$, written $\mathcal{T} \models \varphi$, if $L(\mathcal{T}) \subseteq L(\varphi)$. Given an LTL formula (resp. ω-automaton) φ, the *realizability problem* asks if there exists a transducer $\mathcal{T}$ with input alphabet $\mathcal{I}$ and output alphabet $\mathcal{O}$ such that $\mathcal{T} \models \varphi$. The specification $L(\varphi)$ is *Moore realizable* if such a Moore transducer $\mathcal{T}$ exists, and *Mealy realizable* if such a Mealy transducer $\mathcal{T}$ exists. Note that for an LTL formula, the specification $L(\varphi)$ is Mealy realizable iff $L(\varphi')$ is Moore realizable, where the LTL formula φ' is obtained from φ by replacing all occurrences of $o \in O$ by $\bigcirc o$. The process of constructing a suitable transducer $\mathcal{T}$ is called *synthesis*. The synthesis problem can be solved by computing winning strategies in graph games.

Graph games. We consider two classes of turn-based games on graphs, namely, two-player probabilistic games and two-player deterministic games. The probabilistic games are not needed for synthesis, but we will use them for constructing environment assumptions. For a finite set A, a probability distribution on A is a function $\delta\colon A \to [0,1]$ such that $\sum_{a\in A} \delta(a) = 1$. We denote the set of probability distributions on A by $\mathcal{D}(A)$. Given a distribution $\delta \in \mathcal{D}(A)$, we write $\mathrm{Supp}(\delta) = \{x \in A \mid \delta(x) > 0\}$ for the support of δ. A *probabilistic game graph* $G = ((S,E),(S_1,S_2,S_P),\delta)$ consists of a finite directed graph (S,E), a partition (S_1, S_2, S_P) of the set S of states, and a probabilistic transition function $\delta\colon S_P \to \mathcal{D}(S)$. The states in S_1 are *player*-1 states, where player 1 decides the successor state; the states in S_2 are *player*-2 states, where player 2 decides the successor state; and the states in S_P are *probabilistic* states, where the successor state is chosen according to the probabilistic transition function. We require that for all $s \in S_P$ and $t \in S$, we have $(s,t) \in E$ iff $\delta(s)(t) > 0$, and we often write $\delta(s,t)$ for $\delta(s)(t)$. For technical convenience we also require that every state has at least one outgoing edge. Given a set $E' \subseteq E$ of edges, we write $\mathrm{Source}(E')$ for the set $\{s \in S \mid \exists t \in S : (s,t) \in E'\}$ of states that have an outgoing edge in E'. We write $E_1 = E \cap (S_1 \times S)$ and $E_2 = E \cap (S_2 \times S)$ for the sets of player-1 and player-2 edges. *Deterministic game graphs* are the special case of the probabilistic game graphs with $S_P = \emptyset$, that is, the state space is partitioned into player-1 and player-2 states. In such cases we omit S_P and δ in the definition of the game graph.

Plays and strategies. An infinite path, or *play*, of the game graph G is an infinite sequence $\pi = s_0 s_1 s_2 \ldots$ of states such that $(s_k, s_{k+1}) \in E$ for all $k \geq 0$. We write Π for the set of plays, and for a state $s \in S$, we write $\Pi_s \subseteq \Pi$ for the set of plays that start from s. A *strategy* for player 1 is a function $\alpha\colon S^* \cdot S_1 \to S$ that for all finite sequences of states ending in a player-1 state (the sequence represents a prefix of a play), chooses a successor state to extend the play. A strategy must prescribe only available moves, that is, $\alpha(\tau \cdot s) \in E(s)$ for all $\tau \in S^*$ and $s \in S_1$. The strategies for player 2 are defined analogously. Note that we have only pure (i.e., nonprobabilistic) strategies, but all our results hold even if strategies were probabilistic. We denote by $\mathcal{A}$ and $\mathcal{B}$ the sets of strategies for player 1 and player 2, respectively. A strategy α is *memoryless* if it does not depend on the history of the play but only on the current state. A memoryless player-1 strategy can be represented as a function $\alpha\colon S_1 \to S$, and a memoryless player-2 strategy is a function $\beta\colon S_2 \to S$. We denote by $\mathcal{A}^M$ and $\mathcal{B}^M$ the sets of memoryless strategies for player 1 and player 2, respectively.

Once a start state $s \in S$ and strategies $\alpha \in \mathcal{A}$ and $\beta \in \mathcal{B}$ for the two players are fixed, the outcome of the game is a random walk $\pi_s^{\alpha,\beta}$ for which the probabilities of events are well-defined, where an *event* $\mathcal{E} \subseteq \Pi$ is a measurable set of plays. Given strategies α for player 1 and β for player 2, a play $\pi = s_0 s_1 s_2 \ldots$ is *feasible* if for all $k \geq 0$, we have $\alpha(s_0 s_1 \ldots s_k) = s_{k+1}$ if $s_k \in S_1$, and $\beta(s_0 s_1 \ldots s_k) = s_{k+1}$ if $s_k \in S_2$. Given two strategies $\alpha \in \mathcal{A}$ and $\beta \in \mathcal{B}$, and a state $s \in S$, we write $\mathrm{Outcome}(s,\alpha,\beta) \subseteq \Pi_s$ for the set of feasible plays that start from s. Note that for deterministic game graphs, the set $\mathrm{Outcome}(s,\alpha,\beta)$ contains a single play. For a state $s \in S$ and an event $\mathcal{E} \subseteq \Pi$, we write $\Pr_s^{\alpha,\beta}(\mathcal{E})$ for the probability that a play belongs to $\mathcal{E}$ if the game starts from the state s and the two players follow the strategies α and β.

Objectives. An *objective* for a player is a set $\Phi \subseteq \Pi$ of winning plays. We consider ω-regular sets of winning plays, which are measurable. For a play $\pi = s_0 s_1 s_2 \ldots$, let $\mathrm{Inf}(\pi)$ be the set $\{s \in S \mid s = s_k \text{ for infinitely many } k \geq 0\}$ of states that appear infinitely often in π. We consider safety, Büchi, and parity objectives. Given a set $F \subseteq S$ of states, the *safety objective* $\mathrm{Safe}(F) = \{s_0 s_1 s_2 \ldots \in \Pi \mid \forall k \geq 0 : s_k \in F\}$ requires that only states in F be visited. The *Büchi objective* $\mathrm{Buchi}(F) = \{\pi \in \Pi \mid \mathrm{Inf}(\pi) \cap F \neq \emptyset\}$ requires that some state in F be visited infinitely often. Given a function $p\colon S \to \{0, 1, 2, \ldots, d-1\}$ that maps every state to a *priority*, the *parity objective* $\mathrm{Parity}(p)$ requires that of the states that are visited infinitely often, the least priority be even. Formally, the set of winning plays is $\mathrm{Parity}(p) = \{\pi \in \Pi \mid \min\{p(\mathrm{Inf}(\pi))\} \text{ is even}\}$. Büchi objectives are special cases of parity objectives with two priorities.

Sure and almost-sure winning. Given an objective Φ, a strategy $\alpha \in \mathcal{A}$ is *sure winning* for player 1 from a state $s \in S$ if for every strategy $\beta \in \mathcal{B}$ for player 2, we have $\mathrm{Outcome}(s, \alpha, \beta) \subseteq \Phi$. The strategy α is *almost-sure winning* for player 1 from s for Φ if for every player-2 strategy β, we have $\Pr_s^{\alpha,\beta}(\Phi) = 1$. The sure and almost-sure winning strategies for player 2 are defined analogously. Given an objective Φ, the *sure (resp. almost-sure) winning set* $\langle\langle 1 \rangle\rangle_{sure}(\Phi)$ (resp. $\langle\langle 1 \rangle\rangle_{almost}(\Phi)$) for player 1 is the set of states from which player 1 has a sure (resp. almost-sure) winning strategy. The winning sets $\langle\langle 2 \rangle\rangle_{sure}(\Phi)$ and $\langle\langle 2 \rangle\rangle_{almost}(\Phi)$ for player 2 are defined analogously. It follows from the definitions that for all probabilistic game graphs and all objectives Φ, we have $\langle\langle 1 \rangle\rangle_{sure}(\Phi) \subseteq \langle\langle 1 \rangle\rangle_{almost}(\Phi)$. In general the subset inclusion relation is strict. For deterministic games the notions of sure and almost-sure winning coincide [15], i.e., we have $\langle\langle 1 \rangle\rangle_{sure}(\Phi) = \langle\langle 1 \rangle\rangle_{almost}(\Phi)$, and in such cases we often omit the subscript. Given an objective Φ, the *cooperative winning set* $\langle\langle 1, 2 \rangle\rangle_{sure}(\Phi)$ is the set of states s for which there exist a player-1 strategy α and a player-2 strategy β such that $\mathrm{Outcome}(s, \alpha, \beta) \subseteq \Phi$.

Theorem 1 (Deterministic games [10]). *For all deterministic game graphs and parity objectives Φ, the following assertions hold: (i) $\langle\langle 1 \rangle\rangle_{sure}(\Phi) = S \setminus \langle\langle 2 \rangle\rangle_{sure}(\Pi \setminus \Phi)$; (ii) memoryless sure winning strategies exist for both players from their sure winning sets; and (iii) given a state $s \in S$, if $s \in \langle\langle 1 \rangle\rangle_{sure}(\Phi)$ can be decided in NP $\cap$ coNP.*

Theorem 2 (Probabilistic games [9]). *Given a probabilistic game graph $G = ((S, E), (S_1, S_2, S_P), \delta)$ and a parity objective Φ with d priorities, we can construct a deterministic game graph $\widehat{G} = ((\widehat{S}, \widehat{E}), (\widehat{S_1}, \widehat{S_2}))$ with $S \subseteq \widehat{S}$, and a parity objective $\widehat{\Phi}$ with $d+1$ priorities such that (i) $|\widehat{S}| = O(|S| \cdot d)$ and $|\widehat{E}| = O(|E| \cdot d)$; and (ii) the set $\langle\langle 1 \rangle\rangle_{almost}(\Phi)$ in G is equal to the set $\langle\langle 1 \rangle\rangle_{sure}(\widehat{\Phi}) \cap S$ in $\widehat{G}$. Moreover, memoryless almost-sure winning strategies exist for both players from their almost-sure winning sets in G.*

Realizability games. The realizability problem has the following game-theoretic formulation.

Theorem 3 (Reactive synthesis [17]). *Given an LTL formula or ω-automaton φ, we can construct a deterministic game graph G, a state s_I of G, and a parity objective Φ such that $L(\varphi)$ is Moore (resp. Mealy) realizable iff $s_I \in \langle\langle 1 \rangle\rangle_{sure}(\Phi)$.*

The deterministic game graph G with parity objective Φ referred to in Theorem 3 is called the *Mealy* (resp. *Moore*) *synthesis game* for φ. Starting from an LTL formula φ, we construct the synthesis games by first building a nondeterministic Büchi automaton that accepts $L(\varphi)$ [20]. Then, following the algorithm of [16], we translate this automaton to a deterministic parity automaton that accepts $L(\varphi)$. By splitting every state of the parity automaton w.r.t. inputs I and outputs O, we obtain the Mealy (resp. Moore) synthesis game. Both steps involve exponential blow-ups that are unavoidable: for LTL formulae φ, the realizability problem is 2EXPTIME-complete [18].

Synthesis games, by relating paths in the game graph to the specification $L(\varphi)$, have the following special form. A *Moore synthesis game* $\mathcal{G}$ is a tuple (G, s_I, λ, Φ), where $G = ((S, E), (S_1, S_2))$ is a deterministic bipartite game graph, in which player-1 and player-2 states strictly alternate (i.e., $E \subseteq (S_1 \times S_2) \cup (S_2 \times S_1)$), the initial state $s_I \in S_1$ is a player-1 state, the labeling function $\lambda\colon S \to \mathcal{O} \cup \mathcal{I}$ maps player-1 and player-2 states to letters in $\mathcal{I}$ and $\mathcal{O}$, respectively (i.e., $\lambda(s) \in \mathcal{I}$ for all $s \in S_1$, and $\lambda(s) \in \mathcal{O}$ for all $s \in S_2$), and Φ is a parity objective. Furthermore, synthesis games are deterministic w.r.t. input and output labels, that is, for all edges $(s, s'), (s, s'') \in E$, if $\lambda(s') = \lambda(s'')$, then $s' = s''$. Without loss of generality, we assume that synthesis games are complete w.r.t. input and output labels, that is, for all states $s \in S_1$ (resp. S_2) and $l \in \mathcal{O}$ (resp. $\mathcal{I}$), there exists an edge $(s, s') \in E$ such that $\lambda(s') = l$. We define a function $w\colon \Pi \to \Sigma^\omega$ that maps each play to an infinite word such that $w_i = \lambda(\pi_{2i+1}) \cup \lambda(\pi_{2i+2})$ for all $i \geq 0$. Note that we ignore the label of the initial state.

Given the Moore synthesis game $\mathcal{G}$ for a specification formula or automaton φ (as referred to by Theorem 3), every Moore transducer $\mathcal{T} = (Q, q_I, \Delta, \kappa)$ that satisfies $L(\varphi)$ represents a winning strategy α for player 1 as follows: for all state sequences $\tau \in (S_1 \cdot S_2)^* \cdot S_1$, let w be the finite word such that $w_i = \lambda(\tau_{i+1})$ for all $0 \leq i < |\tau|$; then, if there is an edge $(\tau_{|\tau|}, s') \in E$ with $\lambda(s') = \kappa(\Delta(q_I, \mathrm{odd}(w)))$, let $\alpha(\tau) = s'$, and else let $\alpha(\tau)$ be arbitrary. Conversely, every memoryless winning strategy α of player 1 represents a Moore transducer $\mathcal{T} = (Q, q_I, \Delta, \kappa)$ that satisfies $L(\varphi)$ as follows: let $Q = S_1$, $q_I = s_I$, $\kappa(q) = \lambda(\alpha(q))$, and $\Delta(q, l) = s'$ if $\lambda(s') = l$ and $(\alpha(q), s') \in E$. The construction of a *Mealy synthesis game* for the Mealy realizability problem is similar.

3 Assumptions

We illustrate the difficulties in defining desirable conditions on environment assumptions through several examples. W.l.o.g. we model open reactive systems as Moore transducers, and correspondingly, their environments as Mealy transducers (with inputs and outputs swapped). Given a specification formula or automaton φ that describes the desired behavior of a system $\mathcal{S}$ (a Moore transducer), we search for an assumption on the environment of $\mathcal{S}$ which is sufficient to ensure that $\mathcal{S}$ exists and satisfies $L(\varphi)$. Formally, a language $K \subseteq \Sigma^\omega$ is a *sufficient assumption* for a specification $L \subseteq \Sigma^\omega$ if $(\Sigma^\omega \setminus K) \cup L$ is Moore realizable. In other words, if the specification is given by an LTL formula φ, and the environment assumption by another LTL formula ψ, then ψ is sufficient for φ iff $L(\psi \to \varphi)$ is realizable. In this case, we can view the formula $\psi \to \varphi$ as defining a corrected specification.

Example 1. Consider the specification $\varphi = \mathtt{b}\,\mathsf{U}\,\mathtt{a}$, where U is the *until* operator of LTL. No system $\mathcal{S}$ with input $\mathtt{a}$ and output $\mathtt{b}$ can implement φ, because $\mathcal{S}$ does not control $\mathtt{a}$, and φ is satisfied only if $\mathtt{a}$ eventually is true. We have to weaken the specification to make it realizable. A candidate for the assumption ψ is $\Diamond\mathtt{a}$, because it forces the environment to assert the signal $\mathtt{a}$ eventually. Further candidates are false, which makes the specification trivially realizable, $\Diamond\mathtt{b}$, and $\Diamond\neg\mathtt{b}$, which lead to corrected specifications such as $\varphi' = \Diamond\mathtt{b} \rightarrow \varphi = (\Box\neg\mathtt{b}) \vee \varphi$. The system can implement φ' independent of φ simply by keeping $\mathtt{b}$ low all the time.

Example 1 shows that there may be several different sufficient assumptions for a given specification $L \subseteq \Sigma^\omega$, but not all of them are satisfactory. For instance, the assumption false does not provide the desired information, and the assumption that $\Diamond\mathtt{b}$ cannot be satisfied by any environment that controls only $\mathtt{a}$. Environment assumptions that are unsatisfiable or falsifiable by the system correspond to a corrected specification $\psi \rightarrow \varphi$ that can be satisfied vacuously [2,14] by the system. In order to exclude such assumptions, we require that an environment assumption $K \subseteq \Sigma^\omega$ for L fulfill the following condition.

(1) Realizability by the environment: There exists an implementation of the environment that satisfies K. Formally, we require that the language K be Mealy realizable with input alphabet $\mathcal{O}$ and output alphabet $\mathcal{I}$.

Note that Condition 1 implies that the specification L has to be nonempty for a suitable assumption K to exist. If a formula φ is not satisfiable, then there exists only the trivial solution $\psi = \mathsf{false}$. We assume from now on that specifications are nonempty (i.e., satisfiable). Apart from Condition 1, we aim to restrict the environment "as little as possible." For this purpose, we need to order different assumptions. An obvious candidate for this order is language inclusion.

(2) Minimality: There exists no other sufficient assumption that is realizable by the environment and strictly weaker than K. Formally, there is no language $K' \subseteq \Sigma^\omega$ such that $K \subset K'$ and K' is both a sufficient assumption for L and realizable by the environment.

The following example shows we cannot ask for a *unique* minimal assumption.

Example 2. Consider the specification $\varphi = (\mathtt{b}\,\mathsf{U}\,\mathtt{a}_1) \vee (\neg\mathtt{b}\,\mathsf{U}\,\mathtt{a}_2)$, where $\mathtt{a}_1$ and $\mathtt{a}_2$ are inputs and $\mathtt{b}$ is an output. Again, φ is not realizable. Consider the assumptions $\psi_1 = \Diamond\mathtt{a}_1$ and $\psi_2 = \Diamond\mathtt{a}_2$. Both are sufficient because, assuming ψ_1, the system can keep the signal $\mathtt{b}$ constantly high, and assuming ψ_2, it can keep $\mathtt{b}$ constantly low. Both the assumptions are also realizable by the environment. However, if we assume the disjunction $\psi = \psi_1 \vee \psi_2$, then the system does not know which of the two signals $\mathtt{a}_1$ and $\mathtt{a}_2$ the environment is going to assert eventually. Since a unique minimal assumption has to subsume all other sufficient assumptions and ψ is not sufficient, it follows that there exists no unique minimal assumption that is sufficient.

Let us consider another example to illustrate further difficulties that arise when comparing environment assumptions w.r.t. language inclusion.

Example 3. Consider the specification $\varphi = \square(\mathtt{a} \rightarrow \bigcirc\mathtt{b}) \wedge \square(\mathtt{b} \rightarrow \bigcirc\neg\mathtt{b})$ with input a and output b. The specification is not realizable because whenever a is set to true in two consecutive steps, the system cannot produce a value for b such that φ is satisfied. One natural assumption is $\psi = \square(\mathtt{a} \rightarrow \bigcirc\neg\mathtt{a})$. Another assumption is $\psi' = \psi \vee \Diamond(\neg\mathtt{a} \wedge \bigcirc\mathtt{b})$, which is weaker than ψ w.r.t. language inclusion and still sufficient and realizable by the environment. Looking at the resulting corrected system specification $\psi' \rightarrow \varphi = (\psi \vee \Diamond(\neg\mathtt{a} \wedge \bigcirc\mathtt{b})) \rightarrow \varphi = \psi \rightarrow (\square(\neg\mathtt{a} \rightarrow \bigcirc\neg\mathtt{b}) \wedge \varphi)$, we see that ψ' restricts the system instead of the environment.

Intuitively, using language inclusion as ordering criterion results in minimal environment assumptions that allow only a single implementation for the system. We aim for an assumption that does not restrict the system if possible. One may argue that ψ should refer only to input signals. Let us consider the specification of Example 3 once more. Another sufficient assumption is $\psi'' = (\mathtt{a} \rightarrow \bigcirc\neg\mathtt{a})\,\mathsf{W}(\mathtt{b} \wedge \bigcirc\mathtt{b})$, which is weaker than ψ. This assumption requires that the environment guarantees $\mathtt{a} \rightarrow \bigcirc\neg\mathtt{a}$ as long as the system does not make a mistake (by setting b to true in two consecutive steps), which clearly meets the intuition of an environment assumption. The challenge is to find an assumption that (a) is sufficient, (b) does not restrict the system, and (c) gives the environment maximal freedom.

Note that the assumptions ψ and ψ'' are safety assumptions, while the assumptions in Example 2 are liveness assumptions. In general, every ω-regular language can be decomposed into a safety and a liveness component. We use this separation to provide a way to compute environment assumptions in two steps. In both steps, we restrict the environment strategies of synthesis games to find sufficient environment assumptions. More precisely, we put restrictions on the player-2 edges, which represent decisions made by the environment. If the given specification is satisfiable, then these restrictions lead to assumptions that are realizable by the environment.

4 Safety Assumptions

We first compute assumptions that restrict the safety behavior of the environment.

Nonrestrictive safety assumptions on games. Given a deterministic game graph $G = ((S, E), (S_1, S_2))$, a *safety assumption* is a set $E_S \subseteq E_2$ of player-2 edges requiring that player 2 chooses only edges *not* in E_S. A natural order on safety assumptions is the number of edges in a safety assumption. We write $E_S \leq E_S{}'$ if $|E_S| \leq |E_S{}'|$ holds. For a given player-1 objective Φ, a safety assumption refers to the safety component of the objective, namely, $\Phi_S = \mathrm{Safe}(\langle\langle 1, 2\rangle\rangle_{sure}(\Phi))$. Let $\mathsf{AssumeSafe}(E_S, \Phi) = \{\pi = s_0 s_1 s_2 \ldots \mid$ either (i) there exists $i \geq 0$ such that $(s_i, s_{i+1}) \in E_S$, or (ii) $\pi \in \Phi_S\}$ be the set of all plays in which either one of the edges in E_S is chosen, or that satisfy the safety component of Φ. The safety assumption E_S is *safe-sufficient* for a state $s \in S$ and player-1 objective Φ if player 1 has a winning strategy from s for the modified objective $\mathsf{AssumeSafe}(E_S, \Phi)$. A synthesis game $\mathcal{G} = (G, s_I, \lambda, \Phi)$ with a safety assumption E_S specifies the environment assumption $K(E_S)$ defined as the set of words $w \in \Sigma^\omega$ such that there exists a play $\pi \in \Pi_{s_I}$ with $w = w(\pi)$ and $(\pi_i, \pi_{i+1}) \notin E_S$ for all $i \geq 0$.

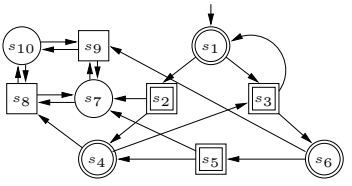


Fig. 1. Game with two equally small safe-sufficient safety assumptions for s_1: $E_S = \{(s_3, s_1)\}$ and $E_S{}' = \{(s_5, s_7)\}$

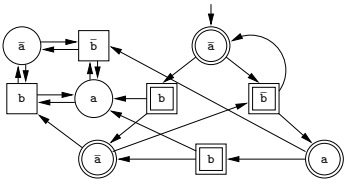


Fig. 2. Synthesis game for $\square(\mathtt{a} \rightarrow \bigcirc\mathtt{b}) \wedge \square(\mathtt{b} \rightarrow \bigcirc\neg\mathtt{b})$

Theorem 4. *Let $\mathcal{G}_\varphi = (G, s_I, \lambda, \Phi)$ be the Moore synthesis game for an LTL formula (or ω-automaton) φ, and let E_S be a safety assumption. If E_S is safe-sufficient for the state s_I and objective Φ, then $K(E_S)$ is a sufficient assumption for the specification* $\mathrm{safe}(\mathrm{pref}(L(\varphi)))$.

The following example shows that there exist safety games without a unique minimal safety assumption that is safe-sufficient.

Example 4. Consider the game shown in Figure 1. Circles denote states of player 1; boxes denote states of player 2. The objective for player 1 is to stay inside the set $\{s_1, \ldots, s_6\}$ of states marked by double lines. Player 1 has no winning strategy from s_1. There are two equally small safety assumptions that are safe-sufficient for s_1: $E_S = \{(s_3, s_1)\}$ and $E_S{}' = \{(s_5, s_7)\}$. In both cases, player 1 has a winning strategy from s_1. If we consider a specification where the corresponding synthesis game has this structure, then neither of these assumptions are satisfactory. Figure 2 shows such a synthesis game, for the specification $\square(\mathtt{a} \rightarrow \bigcirc\mathtt{b}) \wedge \square(\mathtt{b} \rightarrow \bigcirc\neg\mathtt{b})$ with input a and output b (cf. Example 3). Using the safety assumption E_S, the corrected specification would allow only the implementation that keeps b constantly low. The other safety assumption $E_S{}'$ leads to a corrected specification that additionally enforces $\square(\neg\mathtt{a} \rightarrow \bigcirc\neg\mathtt{b})$.

Therefore, besides safe-sufficiency, we look for a safety assumption that does not restrict player 1. This condition can be formalized as follows. Given a deterministic game graph $G = ((S, E), (S_1, S_2))$, a safety assumption E_S is *restrictive* for a state $s \in S$ and a player-1 objective Φ if there exist strategies $\alpha \in \mathcal{A}$ and $\beta \in \mathcal{B}$ for the two players such that the play $\mathrm{Outcome}(s, \alpha, \beta)$ contains an edge from E_S and is in Φ_S. Intuitively, a nonrestrictive safety assumption allows all edges that do not lead to an immediate violation of the safety component of the objective for player 1.

Theorem 5. *Given a deterministic game graph $G = ((S, E), (S_1, S_2))$, an objective Φ for player 1, and a state $s \in S$, if $s \in \langle\langle 1, 2\rangle\rangle_{sure}(\Phi)$, then there exists a unique minimal safety assumption E_S that is nonrestrictive and safe-sufficient for s and Φ. Moreover, if $s \in \langle\langle 1, 2\rangle\rangle_{sure}(\Phi)$ and E_S is the minimal safety assumption for s and Φ, then player 2 has a winning strategy from s for the objective to avoid all edges in E_S.*

Applying Theorem 5 to environment assumptions, we obtain Theorem 6.

Theorem 6. *Let $\mathcal{G} = (G, s_I, \lambda, \Phi)$ be the Moore synthesis game for a satisfiable LTL formula (or ω-automaton) φ. Then there exists a unique minimal safety assumption E_S that is nonrestrictive and safe-sufficient for the state s_I and objective Φ. Moreover, the corresponding assumption $K(E_S)$ is realizable by the environment.*

Computing nonrestrictive safety assumptions. Given a deterministic game graph G and player-1 objective Φ, we compute the unique minimal nonrestrictive and safe-sufficient safety assumption E_S as follows. First, we compute the set $\langle\langle 1,2\rangle\rangle_{sure}(\Phi)$ of states. Note that for this set the players cooperate. We can compute $\langle\langle 1,2\rangle\rangle_{sure}(\Phi)$ in polynomial time for all objectives we consider. In particular, if Φ is a parity objective, then $\langle\langle 1,2\rangle\rangle_{sure}(\Phi)$ can be computed by reduction to Büchi automata [13]. Then the safety assumption E_S is the set of all player-2 edges $(s,t) \in E_2$ such that $s \in \langle\langle 1,2\rangle\rangle_{sure}(\Phi)$ and $t \notin \langle\langle 1,2\rangle\rangle_{sure}(\Phi)$.

Theorem 7. *For every deterministic game graph G and player-1 objective Φ, the edge set $E_S = \{(s,t) \in E_2 \mid s \in \langle\langle 1,2\rangle\rangle_{sure}(\Phi)$ and $t \notin \langle\langle 1,2\rangle\rangle_{sure}(\Phi)\}$ is the unique minimal safety assumption that is nonrestrictive and safe-sufficient for all states $s \in \langle\langle 1,2\rangle\rangle_{sure}(\Phi)$. The set E_S can be computed in polynomial time for parity objectives Φ.*

For the game show in Figure 1, we obtain the safety assumption $E_S = \{(s_3, s_1), (s_5, s_7)\}$. For the corresponding synthesis game in Figure 2, the set E_S defines the environment assumption $\psi_{E_S} = (\neg \mathsf{a} \vee \neg \mathsf{b})\,\mathsf{W}((\neg \mathsf{a} \vee \neg \mathsf{b}) \wedge \mathsf{a} \wedge (\bigcirc \neg \mathsf{b}) \wedge \mathsf{b} \wedge \bigcirc \mathsf{b})$. This safety assumption meets our intuition of a minimal environment assumption, because it states that the environment has to ensure that either a or b is low as long as the system makes no obvious fault by either violating $\Box(\mathsf{a} \rightarrow \bigcirc \mathsf{b})$ or $\Box(\mathsf{b} \rightarrow \bigcirc \neg \mathsf{b})$.

5 Liveness Assumptions

In a second step, we now put liveness assumptions on the environment.

Strongly fair assumptions on games. Given a deterministic game graph $G = ((S,E),(S_1,S_2))$ and a player-1 objective Φ, a *strongly fair assumption* is a set $E_L \subseteq E_2$ of player-2 edges requiring that player 2 plays such that if a state $s \in \mathrm{Source}(E_L)$ is visited infinitely often, then for all states $t \in S$ such that $(s,t) \in E_L$, the edge (s,t) is chosen infinitely often. Let $\mathsf{AssumeFair}(E_L, \Phi)$ be the set of plays π such that either (i) there is a state $s \in \mathrm{Source}(E_L)$ that appears infinitely often in π and there is an edge $(s,t) \in E_L$ that appears only finitely often in π, or (ii) π belongs to the objective Φ. Formally, $\mathsf{AssumeFair}(E_L, \Phi) = \{\pi = s_0 s_1 s_2 \ldots \mid$ either (i) $\exists (s,t) \in E_L$ such that $s_k = s$ for infinitely many k's and there are only finitely many j's such that $s_j = s$ and $s_{j+1} = t$, or (ii) $\pi \in \Phi\}$. The strongly fair assumption $E_L \subseteq E_2$ is *live-sufficient* for a state $s \in S$ and player-1 objective Φ if player 1 has a winning strategy from s for the modified objective $\mathsf{AssumeFair}(E_L, \Phi)$. A state $s \in S$ is *live for player 1* if player 1 has a winning strategy from s for the objective $\mathrm{Safe}(\langle\langle 1,2\rangle\rangle_{sure}(\Phi))$.

Theorem 8. *Given a deterministic game graph $G = ((S,E),(S_1,S_2))$ and a safety or Büchi objective Φ, for every state $s \in S$ that is live for player 1, there exists a strongly fair assumption E_L that is live-sufficient for s and Φ.*

A synthesis game $\mathcal{G} = (G, s_I, \lambda, \Phi)$ with a strongly fair assumption E_L specifies the environment assumption $K(E_L)$ defined as the set of words $w \in \Sigma^\omega$ such that there

exists a play $\pi \in \Pi_{s_I}$ with $w = w(\pi)$ and for all edges $(s,t) \in E_L$, either there exists $i \geq 0$ such that for all $j > i$ we have $\pi_i \neq s$, or there exist infinitely many k's such that $\pi_k = s$ and $\pi_{k+1} = t$. Note that this definition and the structure of synthesis games ensure that $K(E_L)$ is realizable by the environment. These definitions together with Theorem 3 and 8 lead to the following theorem.

Theorem 9. *Let $\mathcal{G} = (G, s_I, \lambda, \Phi)$ be a Moore synthesis game for an LTL formula (or ω-automaton) φ, and let E_L be a strongly fair assumption. If E_L is live-sufficient for the state s_I and objective Φ, then $K(E_L)$ is a sufficient assumption for the specification $L(\varphi)$. Moreover, the assumption $K(E_L)$ is realizable by the environment. Conversely, if Φ is a safety or Büchi objective, if s_I is live for player 1, and if there exists some sufficient assumption $K \neq \emptyset$ for the specification $L(\varphi)$, then there exists a strongly fair assumption that is live-sufficient.*

Computing strongly fair assumptions. We now focus on solution of deterministic player games with objectives $\mathsf{AssumeFair}(E_L, \Phi)$, where Φ is a parity objective. Given a deterministic game graph G, an objective Φ, and a strongly fair assumption E_L on edges, we first observe that the objective $\mathsf{AssumeFair}(E_L, \Phi)$ can be expressed as an implication: a strong fairness condition implies Φ. Hence given Φ as a Büchi or a parity objective, the solution of games with objective $\mathsf{AssumeFair}(E_L, \Phi)$ can be reduced to deterministic Rabin games. However, since deterministic Rabin games are NP-complete we would obtain NP solution (i.e., an NP upper bound), even for the case when Φ is a Büchi objective. We now present an efficient reduction to probabilistic games and show that we can solve deterministic games with objectives $\mathsf{AssumeFair}(E_L, \Phi)$ in NP $\cap$ coNP for parity objectives Φ, and if Φ is a Büchi objective, then the solution is achieved in polynomial time.

Reduction. Given a deterministic game graph $G = ((S, E), (S_1, S_2))$, a parity function p, and a set $E_L \subseteq E_2$ of player-2 edges we construct a probabilistic game $\widetilde{G} = ((\widetilde{S}, \widetilde{E}), (\widetilde{S_1}, \widetilde{S_2}, \widetilde{S_P}), \widetilde{\delta})$ with parity function $\widetilde{p}$ as follows.

1. *State space.* $\widetilde{S} = S \cup \{\widetilde{s} \mid s \in \mathrm{Source}(E_L) \text{ and } E(s) \setminus E_L \neq \emptyset\}$.
2. *State space partition.* $\widetilde{S_1} = S_1$, $\widetilde{S_P} = \mathrm{Source}(E_L)$, and $\widetilde{S_2} = \widetilde{S} \setminus (\widetilde{S_1} \cup \widetilde{S_P})$.
3. *Edges and transition.* We explain edges for the three different kind of states.
   (a) For a state $s \in \widetilde{S_1}$ we have $\widetilde{E}(s) = E(s)$.
   (b) For a state $s \in \widetilde{S_2}$ if $s \in S_2$, then $\widetilde{E}(s) = E(s)$; else $s = \widetilde{s'}$ and $s' \in \mathrm{Source}(E_L)$ and we have $\widetilde{E}(s) = E(s') \setminus E_L$.
   (c) For a state $s \in \widetilde{S_P}$, if $E(s) \subseteq E_L$, then $\widetilde{E}(s) = E(s)$ else $\widetilde{E}(s) = E(s) \cup \{\widetilde{s}\}$. In both case the transition function is uniform over its successors.
4. *Objective.* For all states $s \in S$, we have that $\widetilde{p}(s) = p(s)$, and for a state $\widetilde{s}$ in $\widetilde{S} \setminus S$, let $\widetilde{s}$ be the copy of s, then $\widetilde{p}(\widetilde{s}) = p(s)$.

Intuitively, the edges and transition function can be described as follows: all states s in $\mathrm{Source}(E_L)$ are converted to probabilistic states, and from s all edges in $E(s)$ and the edge to $\widetilde{s}$, which is a copy of s, are chosen uniformly at random. From $\widetilde{s}$ player 2 has the choice of the edges in $E(s)$ that are not contained in E_L.

We refer to the above reduction as the edge assumption reduction and denote it by AssRed, i.e., $(\widetilde{G}, \widetilde{p}) = \mathsf{AssRed}(G, E_L, p)$. The following theorem states the connection

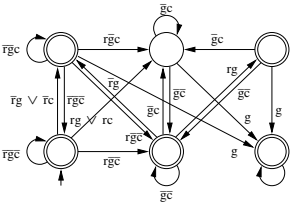


Fig. 3. Constructed environment assumption for $\square(\mathtt{r} \rightarrow \Diamond \mathtt{g}) \wedge \square(\mathtt{c} \rightarrow \bigcirc \neg \mathtt{g})$

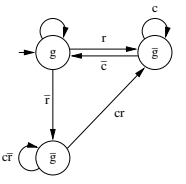


Fig. 4. System constructed with assumption shown in Figure 3

about winning in G for the objective $\mathsf{AssumeFair}(E_L, \mathrm{Parity}(p))$ and winning almost-surely in $\widetilde{G}$ for $\mathrm{Parity}(\widetilde{p})$. The key argument for the proof is as follows. A memoryless almost-sure winning strategy $\widetilde{\alpha}$ in $\widetilde{G}$ can be fixed in G, and it can be shown that the strategy in G is sure winning for the Rabin objective that can be derived from the objective $\mathsf{AssumeFair}(E_L, \mathrm{Parity}(p))$. Conversely, a memoryless sure winning strategy in G for the Rabin objective derived from $\mathsf{AssumeFair}(E_L, \mathrm{Parity}(p))$ can be fixed in $\widetilde{G}$, and it can be shown that the strategy is almost-winning for $\mathrm{Parity}(\widetilde{p})$ in $\widetilde{G}$. A key property useful in the proof is as follows: for a probability distribution μ over a finite set A that assigns positive probability to each element in A, if the probability distribution μ is sampled infinitely many times, then every element in A appears infinitely often with probability 1.

Theorem 10. *Let G be a deterministic game graph, and let Φ be a parity objective defined by a priority function p. Let E_L be a set of player-2 edges, and let $(\widetilde{G}, \widetilde{p}) = \mathsf{AssRed}(G, E_L, p)$. Then $\langle\langle 1 \rangle\rangle_{almost}(\mathrm{Parity}(\widetilde{p})) \cap S = \langle\langle 1 \rangle\rangle_{sure}(\mathsf{AssumeFair}(E_L, \Phi))$.*

Theorem 10 presents a linear-time reduction for $\mathsf{AssumeFair}(E_L, \mathrm{Parity}(p))$ to probabilistic games with parity objectives. Using the reduction of Theorem 2 and the results for deterministic parity games (Theorem 1) we obtain the following corollary.

Corollary 1. *Given a deterministic game graph G, an objective Φ, a set E_L of player-2 edges, and a state s of G, whether $s \in \langle\langle 1 \rangle\rangle_{sure}(\mathsf{AssumeFair}(E_L, \Phi))$ can be decided in quadratic time if Φ is a Büchi objective, and in NP $\cap$ coNP if Φ is a parity objective.*

Complexity of computing a minimal strongly fair assumption. We consider the problem of finding a minimal set of edges on which a strong fair assumption is sufficient. Due to space limitation, we present here only the theorem, the proof can be found in [7].

Theorem 11. *Given a deterministic game graph G, a Büchi objective Φ, a number $k \in \mathbb{N}$, and a state s of G, the problem of deciding if there is a strongly fair assumption E_L with at most k edges (i.e., $|E_L| \leq k$) which is live-sufficient for s and Φ, is NP-hard.*

Computing locally minimal strongly fair assumptions. Since finding a minimal set of edges is NP-hard, we focus on computing a *locally* minimal set of edges. Given a deterministic game graph G, a state s, and a player-1 objective Φ, a set $E_L \subseteq E_2$ of player-2 edges is a *locally-minimal strongly fair assumption* for s and Φ if $s \in \langle\langle 1 \rangle\rangle_{sure}(\mathsf{AssumeFair}(E_L, \Phi))$ and for all proper subsets $E_L{}' \subset E_L$, we

have $s \notin \langle\langle 1 \rangle\rangle_{sure}(\mathsf{AssumeFair}(E_L{}', \Phi))$. A locally-minimal strongly fair assumption $E_L{}^*$ can be computed by a polynomial number of calls to a procedure that checks, given a set E_L of player-2 edges, whether $s \in \langle\langle 1 \rangle\rangle_{sure}(\mathsf{AssumeFair}(E_L, \Phi))$. The computation proceeds as follows. Initially all player-2 edges are in $E_L{}^*$. As long as $s \in \langle\langle 1 \rangle\rangle_{sure}(\mathsf{AssumeFair}(E_L{}^*, \Phi))$, we search for an edge e such that $s \in \langle\langle 1 \rangle\rangle_{sure}(\mathsf{AssumeFair}(E_L{}^* \setminus \{e\}, \Phi))$. If such an e exists, then we remove e from $E_L{}^*$ and proceed; otherwise we stop and return $E_L{}^*$. In the worst case, we have at most $\frac{m \cdot (m+1)}{2}$ procedure calls, where m is the number of player-2 edges.

Theorem 12. *Given a deterministic game graph G, a state s of G, and a parity objective Φ, the computed set $E_L{}^*$ is a locally-minimal strongly fair assumption for s and Φ. If Φ is a Büchi objective, then we can compute a locally-minimal strongly fair assumption $E_L{}^*$ for s and Φ in polynomial time.*

6 Combining Safety and Liveness Assumptions

Now we put everything together. Let φ be an LTL formula (or ω-automaton) and let $\mathcal{G} = (G, s_I, \lambda, \Phi)$ be the corresponding Moore synthesis game. We first compute a nonrestrictive safety assumption E_S as described in Section 4 (Theorem 7). If φ is satisfiable, then it follows from Theorem 6 that E_S exists and that the corresponding environment assumption $K(E_S)$ is realizable by the environment. Then, we modify the player-1 objective with the computed safety assumption: we extend the set of winning plays for player 1 with all plays in which player 2 follows one of the edges in E_S. Since E_S is safe-sufficient for s_I and Φ, it follows that s_I is live for player 1 in the modified game. On the modified game, we compute a locally-minimal strongly fair assumption $E_L{}^*$ as described in Section 5 (Theorem 12). Finally, using Theorems 8 and 9, we conclude the following.

Theorem 13. *Given an LTL formula (or ω-automaton) φ, let $\hat{K} = K(E_S) \cap K(E_L{}^*)$, where E_S and $E_L{}^*$ are computed as described in Theorems 7 and 12. If $\hat{K} \neq \emptyset$, then $\hat{K}$ is a sufficient assumption for the specification $L(\varphi)$ which is realizable by the environment. Conversely, if the Moore synthesis game for φ has a safety or Büchi objective, and if there exists a sufficient assumption $K \neq \emptyset$ for the specification $L(\varphi)$, then the computed assumption $\hat{K}$ is nonempty.*

Recall the example from the introduction with the signals $\mathtt{r}$, $\mathtt{c}$, and $\mathtt{g}$, and the specification $\Box(\mathtt{r} \rightarrow \bigcirc\Diamond\mathtt{g}) \wedge \Box((\mathtt{c} \vee \mathtt{g}) \rightarrow \bigcirc\neg\mathtt{g})$. Our algorithm computes the environment assumption $\widehat{\psi}$ shown in Figure 3 (double lines indicate Büchi states). Since it is not straightforward to describe the language using an LTL formula, we give its relation to the assumptions proposed in the introduction. The computed assumption $\widehat{\psi}$ includes $\psi_1 = \Box\neg\mathtt{c}$ and $\psi_2 = \Box\Diamond\neg\mathtt{c}$, is a strict subset of $\psi_6 = \xi\,\mathsf{W}(\xi \wedge (\mathtt{c} \vee \mathtt{g}) \wedge \bigcirc\mathtt{g})$ with $\xi = \mathtt{r} \rightarrow \bigcirc\Diamond(\neg\mathtt{c} \vee \mathtt{g})$, and is incomparable to all other sufficient assumptions. Even though the computed assumption is not the weakest w.r.t. language inclusion, it still serves its purpose: Figure 4 shows a system synthesized from the corrected specification of [12] using the environment assumption $\widehat{\psi}$.

Acknowledgements. The authors thank Rupak Majumdar for stimulating and fruitful discussions. This research was supported in part by the NSF grants CCR-0132780,

CNS-0720884, and CCR-0225610, by the Swiss National Science Foundation (Indo-Swiss Research Program and NCCR MICS), and by the European COMBEST project.

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2, 117–126 (1987)
2. Beer, I., Ben-David, S., Eisner, U., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 279–290. Springer, Heidelberg (1997)
3. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: from parity games to safety games. Theoretical Informatics and Applications, 261–275 (2002)
4. 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)
5. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: A case study. In: DATE 2007, pp. 1188–1193. ACM, New York (2007)
6. Bobaru, M.G., Pasareanu, C., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: CAV 2008. LNCS. Springer, Heidelberg (accepted for publication, 2008)
7. Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. CoRR, abs/0805.4167 (2008)
8. Chatterjee, K., Henzinger, T.A., Jurdziński, M.: Games with secure equilibria. Theoretical Computer Science 365, 67–82 (2006)
9. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Simple stochastic parity games. In: Baaz, M., Makowsky, J.A. (eds.) CSL 2003. LNCS, vol. 2803, pp. 100–113. Springer, Heidelberg (2003)
10. Emerson, E.A., Jutla, C.: Tree automata, mu-calculus and determinacy. In: FOCS 1991, pp. 368–377. IEEE, Los Alamitos (1991)
11. Faella, M.: Games you cannot win. In: Workshop on Games and Automata for Synthesis and Validation, Lausanne, Switzerland (2007)
12. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD 2006, pp. 117–124. IEEE, Los Alamitos (2006)
13. King, V., Kupferman, O., Vardi, M.Y.: On the complexity of parity word automata. In: FOSSACS 2006, pp. 276–286. Springer, Heidelberg (2001)
14. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 82–96. Springer, Heidelberg (1999)
15. Martin, D.A.: Borel determinacy. Annals of Mathematics, pp. 363–371 (1975)
16. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: LICS 2006, pp. 255–264. IEEE, Los Alamitos (2006)
17. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL 1989, pp. 179–190. ACM, New York (1989)
18. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Weizmann Institute of Science (1992)
19. Specman elite - testbench automation, http://www.verisity.com/products/specman.html
20. Vardi, M., Wolper, P.: Reasoning about infinite computations. Information and Computation, 1–37 (1994)
21. Vera - testbench automation, http://www.synopsys.com/products/vera/vera.html

Smyle: A Tool for Synthesizing Distributed Models from Scenarios by Learning*

Benedikt Bollig[1], Joost-Pieter Katoen[2], Carsten Kern[2], and Martin Leucker[3]

[1] LSV, ENS Cachan, CNRS
[2] RWTH Aachen University
[3] TU München

1 Overview

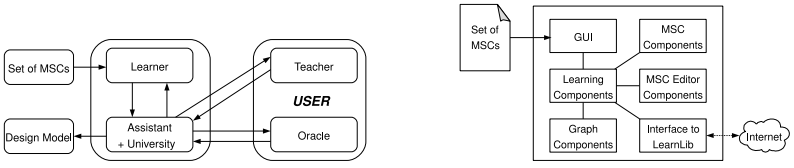
This paper presents *Smyle*, a tool for synthesizing asynchronous and distributed implementation models from sets of scenarios that are given as message sequence charts (MSCs). The latter specify desired or unwanted behavior of the system to be. Provided with such positive and negative example scenarios, *Smyle* employs dedicated learning techniques and *propositional dynamic logic* (PDL) over MSCs to generate a system model that conforms with the given examples.

Synthesizing distributed systems from user-specified scenarios is becoming increasingly en vogue [7]. There exists a wide range of approaches for synthesizing implementation models from a priori given scenarios [5, 6, 8, 11, 14, 15]. The approaches mainly differ in their specification language, the inference procedure, and the final implementation model. Several of them employ MSCs as specification language because they are standardized by the ITU Z.120 [9] and adopted by the UML as sequence diagrams. Other approaches try to utilize more expressive notations like triggered MSCs [14], high-level MSCs [6], or live sequence charts [8]. On the one hand, more expressive power results in richer specifications. On the other hand, however, it is just this great expressiveness that disqualifies them for non-professional or a fortiori unexperienced users, which are overstrained by these formalisms. As requirements specifications over and over demonstrate, human beings strongly prefer to express scenarios in terms of simple pictures including the acting entities and their interaction. Due to this reason, we will restrict to so-called basic MSCs, only, which consist of processes (i.e., vertical axes denoting evolution of time) and messages (i.e., horizontal or slanted arrows between processes signifying asynchronous information exchange).

Many approaches to synthesizing distributed systems typically model synchronous communication, infer labeled transition systems, and use standard automata-theoretic solutions to project the global system onto its local components. As can be shown easily, this results in missing or implied behavior that was not stipulated by the user. In contrast, we regard asynchronous communication behavior and derive a distributed model in terms of a *message passing automaton* (MPA), which models the asynchronous communication in a natural manner deploying FIFO channels. It consist of one local automaton for every process involved in the system. Harel in his recent article [7] states that it is an intrinsically difficult task to

* This work is partially supported by the DAAD (Procope 2008).

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 162–166, 2008.



(a) Learning overview (b) Package overview

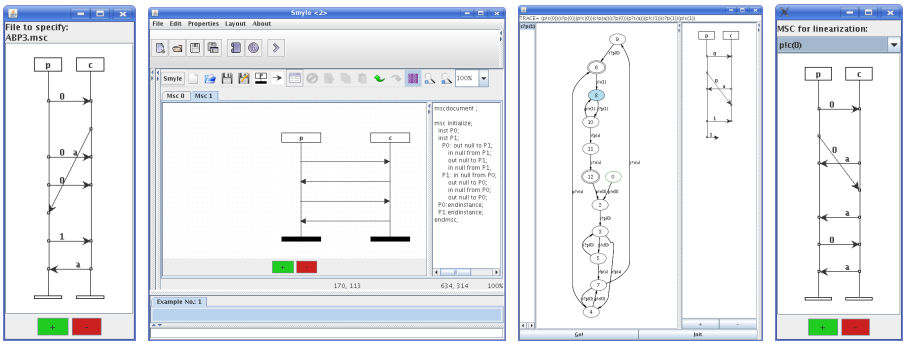
Fig. 1. *Smyle*'s architecture

"[...] distribute the intuitive played-in behavior [...]" and it is still a *dream* to employ *scenario-based programming*. Nevertheless we try to converge to this vision using the tool *Smyle* presented in this paper.

2 *Smyle* from a User Perspective

Smyle is an acronym for *Synthesizing Models bY Learning from Examples*. Its major objective is to ease the development of concurrent systems. More specifically, the overall goal is to derive communication models of concurrent systems. The synthesis process starts by providing the tool with a set of sample MSCs where each MSC is either positive or negative. Positive MSCs describe system behavior that is *possible* and negative MSCs characterize *unwanted* or *forbidden* behavior. *Smyle* focuses on basic-MSC features like asynchronous message exchange and forbids to deploy the complete MSC standard—which allows for alternation, loops etc.—on purpose: the more expressive a specification language gets the less intuitive and manageable it becomes. Simple pictures however are easy to understand and easy to supply. As mentioned in the previous section, human beings prefer to describe system runs by means of simple examples and pictures. Basic MSCs constitute such a device. More information about the formal basis *Smyle* builds on can be found in [3].

The learning chain: In order to initiate the synthesis process, a so-called *learning setup* is specified where the channel capacities of the final system are fixed a priori by a bound $B \in \mathbb{N}$. The user can choose from two variants: she may either want to *learn* a *universally-B-bounded* system, which means that the outcome will be realizable using finite channel capacity B, thus resulting in a finite-state system, or to infer a possibly infinite-state system by requiring existential bounds on the system's channels. An *existentially-B-bounded* learning setup allows the system developer to include system behavior that may exceed the system's channel bound B but at the same time guarantees that there is at least one execution (i.e., a total ordering of events or a *linearization*) that adheres to this limit for each scenario recognized by the final system. Hence, an appropriate scheduler will always be able to execute the *good* linearizations (i.e., runs not exceeding B) and disregard the ones going beyond the bound. Having chosen a learning setup, a set of MSCs has to be provided as initial input to the tool. *Smyle* will ask the user to classify these MSCs and start the learning procedure (cf. Figure 1(a)). Successively, new MSCs as depicted in Figure 2(a)



(a) MSC (b) Integrated MSC editor with MSC to classify (c) Simulation component (d) MSC rejected by φ

Fig. 2. *Smyle* GUI

are presented to the user (acting as Teacher) who in turn has to classify these scenarios as either wanted (positive) or illegal (negative). Whenever *Smyle* has a complete and consistent view of the current internal system, it presents a window (cf. Figure 2(c)) for testing and simulating the derived system. Within this component, the user (now acting as Oracle) may execute actions to see how the system behaves. These actions are monitored and the related scenario is depicted as MSC on the right hand side of the frame (cf. Figure 2(c)). If, after an intensive simulation, there is no evidence for wrong or missing behavior, the user will terminate the simulation session and the concurrent system is deduced. If, however, some illegal or missing behavior is detected, then the user can use the corresponding MSC as counterexample, or, respectively, edit the missing scenario to a (positive) counterexample. This singleton set of counterexamples may of course be enriched by additional MSCs, and the learning procedure continues as explained until reaching the next consistent model. An exemplifying video of this learning process can be downloaded from the tool's webpage [1].

The MSC editor: When new MSCs have to be specified in order to start or continue the learning phase, *Smyle* can either load MSC documents containing basic MSCs from the file system or offer to use the integrated MSC editor (cf. Figure 2(b)) for easy specification of basic MSCs. The MSCs can directly be classified and fed back to *Smyle* in order to derive a new MPA. The editor also provides functionality for storing MSCs in many different formats (e.g., LaTeX, fig, etc.). An extended, stand-alone version of the editor covering the ITU Z.120 standard to a large extend will soon be available [1].

Easing the learning process: In order to simplify the user's task of classifying scenarios, *Smyle* contains means for specifying formulas from PDL over MSCs, a simple logic that comes with an efficiently solvable membership problem [4]. Like MSCs, PDL formulas are used to express desired or illegal behavior, but in a broader sense. They are to be seen as general rules which apply for *all* runs of

the system (and not only all executions of one scenario). Hence, if a user detects generally wanted or unwanted properties of the presented MSCs she may specify formulas which express these *generics*. *Smyle* is supplied with these formulas and can, from that moment on, accept or reject all MSCs that fulfill or, respectively, violate one of these formulas. This technique reduces the number of user queries substantially. An example formula is $\varphi = \mathbf{A}\big([\, p?c(\mathtt{a}); \mathsf{proc}; p?c(\mathtt{a})\,]\ \mathbf{false}\big)$ which states that there must not be two subsequent occurrences of the same action (i.e., $p?c(\mathtt{a})$) on the same process p. Hence, if formula φ is fed to *Smyle* as negative generic, all MSCs featuring this behavior, e.g., the one in Figure 2 (d), would be regarded as negative samples without questioning the user.

3 *Smyle*'s Implementation Details

Smyle is a platform-independent application written in `Java 1.5`. For visualization purposes, e.g., displaying MSCs and the implementation model, it uses the graph-visualization libraries `Grappa` [1] and `JGraph` [2], and employs the MSC2000 parser [12] for parsing the input MSC documents. As depicted in Figure 1(b), *Smyle* consists of six main packages: the graphical user interface (GUI), one package for MSC components, one for learning components, one comprising the MSC editor functionality, one for graph components, and an interface to the learning library *LearnLib* [13]. The functionality of these components is briefly described in the following.

MSC components: This package contains the MSC2000 [12] parser for handling MSC documents according to the ITU Z.120 standard [9]. It provides the classes for representing the internal MSC objects.

Learning components: The tasks of the learning component are manifold. It contains important functionality for efficient partial-order treatment, harbors the simulator which can be applied to the learned model, and comprises the *Learner*, the *Assistant*, and the *University*, which acts as mediator between the components of this package and the other packages as shown in Figure 1(a).

MSC editor components: The MSC editor components feature the implementation of an integrated MSC editor, which is able to load, store, and alter basic MSCs. Moreover, the created MSCs can be exported to LaTeX and the `fig` format and thus can be converted, using available tools, to all other prevalent graphical formats (e.g., eps, pdf, jpeg).

Graph components: The graph components package includes functionality for checking MSC behavior (e.g., the FIFO property) and the consistency of the implementation models.

Interface to LearnLib*:* This package includes an interface to a learning library called *LearnLib* [13], which implements Angluin's algorithm L^* [2]. This interface is by courtesy of the *Fachbereich Informatik, Lehrstuhl 5* (University of Dortmund).

[1] Grappa: http://www.research.att.com/~john/Grappa/
[2] JGraph: http://www.jgraph.com/

4 Future Work

We have applied *Smyle* to a number of examples. We inferred, for example, a model for the ABP, one for a part of the USB 1.1 protocol and one for a leader election protocol for a unidirectional ring. Moreover, *Smyle* has also been considered in [10] where scenarios represented as MSCs are derived from natural language specifications.

For future work, we plan to extend the formula component and integrate it into the MSC editor to be able to derive PDL formulas from visually annotated MSCs. Moreover, we intend to apply *Smyle* to larger sized case studies to evaluate its feasibility for real world problems.

The synthesis tool *Smyle*, the MSC editor as well as dedicated theoretic background information and an exemplifying video presenting the learning chain can be freely downloaded for educational and research purposes from:

`http://www.smyle-tool.org/`

References

1. Smyle webpage, `http://www.smyle-tool.org/`
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
3. Bollig, B., Katoen, J.-P., Carsten, K., Leucker, M.: Replaying play in and play out: Synthesis of design models from scenarios by learning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 435–450. Springer, Heidelberg (2007)
4. Bollig, B., Kuske, D., Meinecke, I.: Propositional dynamic logic for message-passing systems. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 303–315. Springer, Heidelberg (2007)
5. Damas, C., Lambeau, B., Dupont, P.: Generating annotated behavior models from end-user scenarios. IEEE Trans. Softw. Eng. 31(12), 1056–1073 (2005)
6. Genest, B., Muscholl, A., Seidl, H., Zeitoun, M.: Infinite-state high-level mscs: model-checking and realizability. J. Comput. Syst. Sci. 72(4), 617–647 (2006)
7. Harel, D.: Can programming be liberated, period? Computer 41(1), 28–37 (2008)
8. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003)
9. ITU-TS Recommendation Z.120: Message Sequence Chart 1999 (MSC 1999) (1999)
10. Kof, L.: Scenarios: Identifying missing objects and actions by means of computational linguistics. In: 15th IEEE RE, pp. 121–130 (2007)
11. Mäkinen, E., Systä, T.: MAS – An interactive synthesizer to support behavioral modeling in UML. In: ICSE, pp. 15–24. IEEE Computer Society, Los Alamitos (2001)
12. Neukirchen, H.: MSC 2000, Parser (2000), `neukirchen@informatik.unigoettingen.de`
13. Raffelt, H., Steffen, B.: LearnLib: A library for automata learning and experimentation. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 377–380. Springer, Heidelberg (2006)
14. Sengupta, B., Cleaveland, R.: Triggered message sequence charts. IEEE Trans. Softw. Eng. 32(8), 587–607 (2006)
15. Uchitel, S., Brunet, G., Chechik, M.: Behaviour model synthesis from properties and scenarios. In: ICSE, pp. 34–43. IEEE Computer Society, Los Alamitos (2007)

SYCRAFT: A Tool for Synthesizing Distributed Fault-Tolerant Programs*

Borzoo Bonakdarpour and Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48823, U.S.A.
{borzoo,sandeep}@cse.msu.edu

Abstract. We present the tool SYCRAFT (*SYmboliC synthesizeR and Adder of Fault-Tolerance*). In SYCRAFT, a distributed fault-intolerant program is specified in terms of a set of processes and an invariant. Each process is specified as a set of actions in a guarded command language, a set of variables that the process can read, and a set of variables that the process can write. Given a set of fault actions and a specification, the tool transforms the input distributed fault-intolerant program into a distributed fault-tolerant program via a symbolic implementation of respective algorithms.

1 Introduction

Distributed programs are often subject to unanticipated events called *faults* (e.g., message loss, processor crash, etc.) caused by the environment that the program is running in. Since identifying the set of all possible faults that a distributed system is subject to is almost unfeasible at design time, it is desirable for designers of *fault-tolerant* distributed programs to have access to synthesis methods that transform existing fault-intolerant distributed programs to a fault-tolerant version.

Kulkarni and Arora [4] provide solutions for automatic addition of fault-tolerance to fault-intolerant programs. Given an existing program, say p, a set f of faults, a safety condition, and a reachability constraint, their solution synthesizes a fault-tolerant program, say p', such that (1) in the absence of faults, the set of computations of p' is a subset of the set of computations of p, and (2) in the presence of faults, p' never violates its safety condition, and, starting from any state reachable by program and fault transitions, p' satisfies its reachability condition in a finite number of steps. In particular, they show that the synthesis problem in the context of distributed programs is NP-complete in the state space of the given intolerant program.

To cope with this complexity, in a previous work [2], we developed a set of symbolic heuristics for synthesizing moderate-sized fault-tolerant distributed

* This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 167–171, 2008.

programs. The tool SYCRAFT implements these symbolic heuristics. It is written in C++ and the symbolic algorithms are implemented using the Glu/CUDD 2.1 package. The source code of SYCRAFT is available freely and can be downloaded from http://www.cse.msu.edu/~sandeep/sycraft

Related work. Our synthesis problem is in sprit close to controller synthesis and game theory problems. However, there exist several distinctions in theories and, hence, the corresponding tools. In particular, in controller synthesis and game theory, the notion of addition of *recovery* computations does not exist, which is a crucial concept in fault-tolerant systems. Moreover, we model *distribution* by specifying read-write restrictions, whereas related tools and methods (e.g., GASt, Supremica, and the SMT-based method in [3]) do not address the issue of distribution.

2 The Tool SYCRAFT

2.1 Input Language

We illustrate the input language and output format of SYCRAFT using a classic example from the literature of fault-tolerant distributed computing, the Byzantine agreement problem [5] (Figure 1). In Byzantine agreement, the program consists of a *general* g and three (or more) *non-general* processes: 0, 1, and 2. Since, the non-general processes have the same structure, we model them as a process j that ranges over 0..2. The general is not modeled as a process, but by two global variables *bg* and *dg*. Each process maintains a decision *d*; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1 or 2, where the value 2 denotes that the corresponding process has not yet received the value from the general. Each non-general process also maintains a Boolean variable *f* that denotes whether that process has finalized its decision. To represent a Byzantine process, we introduce a variable *b* for each process; if *b.j* is true then process j is Byzantine. In SYCRAFT, a distributed fault-intolerant program comprises of the following sections:

Process sections. Each process includes (1) a set of *process actions* given in *guarded commands*, (2) a *fault section* which accommodates fault actions that the process is subject to, (3) a *prohibited* section which defines a set of transitions that the process is not allowed to execute, (4) a set of variables that the process is allowed to *read*, and (5) a set of variables that the process is allowed to *write*. The syntax of actions is of the form $g \rightarrow st$, where the guard g is a Boolean expression and statement st is a set of (possibly non-deterministic) assignments. The semantics of actions is such that at each time, one of the actions whose guard is evaluated as *true* is chosen to be executed non-deterministically. The read-write restrictions model the issue of distribution in the input program. Note that in SYCRAFT, fault actions are able to read and write all the program variables. Prohibited transitions are specified as a conjunction of an (unprimed) source predicate and a (primed) target predicate.

In the context of Byzantine agreement, each non-general process copies the decision value from the general (Line 6) and then finalizes that value (Line 8). A fault action may turn a process to a Byzantine process, if no other process is Byzantine (Line 10). Faults also change the decision (i.e., variables d and f) of a Byzantine process arbitrarily and non-deterministically (Line 12). In SYCRAFT, one can specify faults that are not associated with a particular process. This feature is useful for cases where faults affect global variables, e.g., the decision of the general (Lines 18-22). In the *prohibited* section, we specify transitions that violate safety by process (and not fault) actions. For instance, it is unacceptable for a process to change its final decision (Line 14). Finally, a non-general process is allowed to read its own and other processes' d values and update its own d and f values (Lines 15-16).

Invariant section. The invariant predicate has a triple role: it (1) is a set of states from where execution of the program satisfies its safety specification (described below) in the absence of faults, (2) must be closed in the execution of the input program and, (3) specifies the *reachability* condition of the program, i.e., if occurrence of faults results in reaching a state outside the invariant, the (synthesized) fault-tolerant program must *safely* reach the invariant predicate in a finite number of steps. In order to increase the chances of successful synthesis, it is desired that SYCRAFT is given the weakest possible invariant. In the context of our example, the following states define the invariant: (1) at most one process may be Byzantine (Line 24), (2) the d value of a non-Byzantine non-general process is either 2 or equal to dg (Line 25), and (3) a non-Byzantine undecided process cannot finalize its decision (Line 26), or, if the general is Byzantine and other processes are non-Byzantine their decisions must be identical and not equal to 2 (Line 28).

Specification section. Our notion of specification is based on the one defined by Alpern and Schneider [1]. The specification section describes the *safety* specification as a set of *bad prefixes* that should not be executed neither by a process nor a fault action. Currently, the size of such prefixes in SYCRAFT is two (i.e., a set of transitions). The syntax of specification section is the same as prohibited section described above. In the context of our example, *agreement* requires that the final decision of two non-Byzantine processes cannot be different (Lines 30-31). And, *validity* requires that if the general is non-Byzantine then the final decision of a non-Byzantine process must be the same as that of the general (Lines 32). Notice that in the presence of a Byzantine process, the program may violate the safety specification.

2.2 Internal Functionality

SYCRAFT implements three nested symbolic fixedpoint computations. The inner fixedpoint deals with computing the set of states reachable by the input intolerant program and fault transitions. The second fixedpoint computation identifies the set *ms* of states from where the safety condition may be violated by fault transitions. It makes *ms* unreachable by removing program transitions that end

```
1)  program Byzantine_Agreement;
2)  const N = 2;
3)  var boolean bg, b.0..N, f.0..N;
4)     int dg: domain 0..1, d.0..N: domain 0..2;
    {-------------------------------------------------------------------------}
5)  process j: 0..N
6)        ((d.j == 2) & !f.j & !b.j) --> d.j := dg;
7)     []
8)        ((d.j != 2) & !f.j & !b.j) --> f.j := true;
9)     fault Byzantine_NonGeneral
10)           (!bg) & (forall p in 0..N::(!b.p)) --> b.j := true;
11)       []
12)           (b.j) --> (d.j := 1) [] (d.j := 0) []
                        (f.j := false) [] (f.j := true);
13)    endfault
14)    prohibited (!b.j)&(!b.j')&(f.j)&((d.j!=d.j') | (!f.j'))
15)    read: d.0..N, dg, f.j, b.j;
16)    write: d.j, f.j;
17) endprocess
    {-------------------------------------------------------------------------}
18) fault Byzantine_General
19)       !bg & (forall p in 0..N:: (!b.p)) --> bg := true;
20)    []
21)       bg --> (dg := 1) [] (dg := 0);
22) endfault
    {-------------------------------------------------------------------------}
23) invariant
24)    (!bg & (forall p, q in 0..N:(p != q):: (!b.p | !b.q))&
25)    (forall r in 0..N::(!b.r => ((d.r == 2) | (d.r == dg)))) &
26)    (forall s in 0..N:: ((!b.s & f.s) => (d.s != 2))))
27)    |
28)    (bg & (forall t in 0..N:: (!b.t)) &
              (forall a,b in 0..N::((d.a==d.b)&(d.a!=2))));
    {-------------------------------------------------------------------------}
29) specification:
30)    (exists p, q in 0..N: (p != q) :: (!b.p' & !b.q' & (d.p' != 2) &
31)       (d.q' != 2) & (d.p' != d.q') & f.p' & f.q'))  |
32)    (exists r in 0..N:: (!bg' & !b.r' & f.r' & (d.r' != 2) & (d.r' != dg')));
```

Fig. 1. The Byzantine agreement problem as input to SYCRAFT

in a state in *ms*. Note that in a distributed program, since processes cannot read and write all variables, each transition is associated with a *group* of transitions. Thus, removal or addition of a transition must be done along with its corresponding group. The outer fixedpoint computation deals with resolving *deadlock* states by either (if possible) adding safe recovery simple paths from deadlock states to the program's invariant predicate, or, making them unreachable via adding minimal restrictions on the program.

2.3 Output Format

The output of SYCRAFT is a fault-tolerant program in terms of its actions. Figure 2 shows the fault-tolerant version of Byzantine agreement with respect to process j = 0. SYCRAFT organizes these actions in three categories. *Unchanged actions* entirely exist in the input program (e.g., action 1). *Revised actions* exist in the input program, but their guard or statement have been strengthened (e.g., Line 8 in Figure 1 and actions 2-5 in Figure 2). SYCRAFT adds *Recovery actions* to

```
-------------------------------------------------------------------------------
UNCHANGED ACTIONS:
-------------------------------------------------------------------------------
1-((d0==2) & !(f0==1)) & !(b0==1)            -->   (d0:=dg)
-------------------------------------------------------------------------------
REVISED ACTIONS:
-------------------------------------------------------------------------------
2-(b0==0) & (d0==1) & (d1==1) & (f0==0)      -->   (f0:=1)
3-(b0==0) & (d0==0) & (d2==0) & (f0==0)      -->   (f0:=1)
4-(b0==0) & (d0==0) & (d1==0) & (f0==0)      -->   (f0:=1)
5-(b0==0) & (d0==1) & (d2==1) & (f0==0)      -->   (f0:=1)
-------------------------------------------------------------------------------
NEW RECOVERY ACTIONS:
-------------------------------------------------------------------------------
6-(b0==0)&(d0==0)&(d1==1)&(d2==1)&(f0==0)    -->   (d0:=1)[]((d0:=1), (f0:=1))
7-(b0==0)&(d0==1)&(d1==0)&(d2==0)&(f0==0)    -->   (d0:=0)[]((d0:=1), (f0:=1))
-------------------------------------------------------------------------------
```

Fig. 2. Fault-tolerant Byzantine agreement

enable the program to safely converge to its invariant predicate. Notice that the strengthened actions prohibit the program to reach a state from where validity or agreement is violated in the presence of faults. It also prohibits the program to reach deadlock states from where safe recovery is not possible.

3 Conclusion

SYCRAFT allows for transformation of moderate-sized fault-intolerant distributed programs to their fault-tolerant version with respect to a set of uncontrollable faults, a safety specification, and a reachability constraint. In addition to the obvious benefits of automated program synthesis, we have observed that SYCRAFT can be potentially used to debug specifications as well, since the algorithms in SYCRAFT tend to add minimal restrictions on the synthesized program. Thus, testing approaches can be used to evaluate behaviors of the synthesized programs to identify missing requirements. To address this potential application of program synthesis, we plan to add *supervised synthesis* features to SYCRAFT.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21, 181–185 (1985)
2. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 3–10 (2007)
3. Finkbeiner, B., Schewe, S.: SMT-based synthesis of distributed systems. In: Automated Formal Methods (AFM) (2007)
4. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), pp. 82–93 (2000)
5. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) 4(3), 382–401 (1982)

Subsequence Invariants*

Klaus Dräger and Bernd Finkbeiner

Universität des Saarlandes
Fachrichtung Informatik, 66123 Saarbrücken, Germany
{draeger,finkbeiner}@cs.uni-sb.de

Abstract. We introduce subsequence invariants, which characterize the behavior of a concurrent system in terms of the occurrences of synchronization events. Unlike state invariants, which refer to the state variables of the system, subsequence invariants are defined over auxiliary counter variables that reflect how often the event sequences from a given set have occurred so far. A subsequence invariant is a linear constraint over the possible counter values. We allow every occurrence of a subsequence to be interleaved arbitrarily with other events. As a result, subsequence invariants are preserved when a given process is composed with additional processes. Subsequence invariants can therefore be computed individually for each process and then be used to reason about the full system. We present an efficient algorithm for the synthesis of subsequence invariants. Our construction can be applied incrementally to obtain a growing set of invariants given a growing set of event sequences.

1 Introduction

An invariant is an assertion that holds true in every reachable state. Since most program properties can either directly be stated as invariants or need invariants as part of their proof, considerable effort has been devoted to synthesizing invariants automatically from the program text [1, 2, 3, 4, 7, 14].

The most natural approach to invariant generation, followed in almost all previous work, is to look for constraints over the program variables that are inductive with respect to the program transitions. This approach works well for sequential programs, but often fails for concurrent systems: to be inductive, the invariants must refer to variables from all (or at least multiple) processes; working on the product state space, however, is only feasible if the number of processes is very small.

We introduce a new type of program invariants, which we call *subsequence invariants*. Instead of referring to program variables, subsequence invariants characterize the behavior of a concurrent system in terms of the occurrences of synchronization events. Subsequence invariants are defined over auxiliary counter variables that reflect how often the event sequences from a given set of subsequences have occurred so far. A subsequence invariant is a linear constraint over

* This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 172–186, 2008.

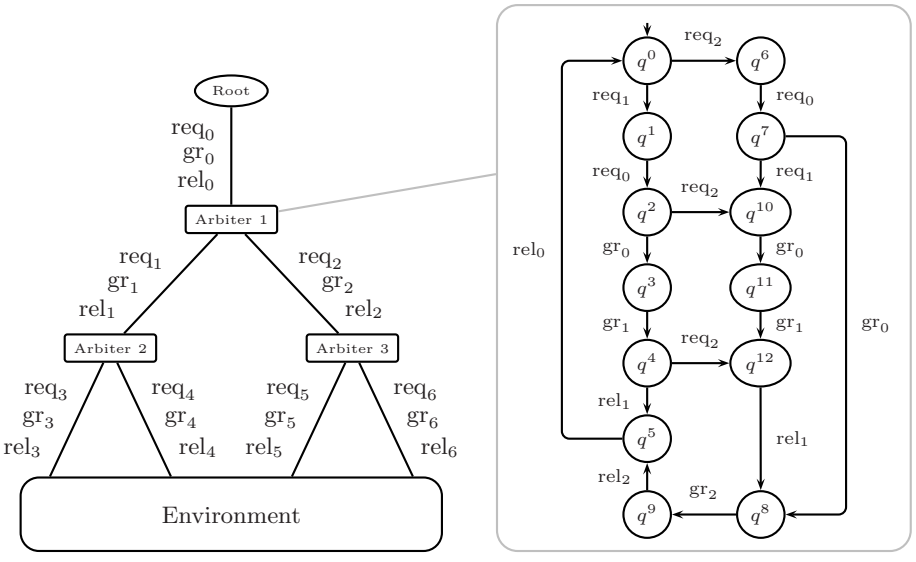


Fig. 1. Arbiter tree: Access to a shared resource is controlled by binary arbiters arranged in a tree, with a central root process

the possible counter values. Each occurrence of a subsequence may be *scattered* over a sequence of synchronization events: for example, the sequence *acacb* contains two occurrences (*acacb* and *acacb*) of the subsequence *ab*. This robustness with respect to arbitrary interleavings with other events ensures that subsequence invariants are preserved when a given process is composed with additional processes. Subsequence invariants can therefore be computed individually for each process and then be used to reason about the full system.

As an example, consider the arbiter tree shown in Figure 1. The environment represents the clients of the system, which may request access to a shared resource from one of the leaf nodes of the arbiter tree. The arbiter node then sends a request to its parent in the tree. This request is forwarded up to the central root process, which generates a grant as soon as the resource is available. The grant is propagated down to a requesting client, which then accesses the resource and eventually sends a release signal when it is done. Each arbiter node satisfies the following subsequence invariants:

(1) Whenever a grant is given to a child, the number of grants given to the other child so far equals the number of releases received from it. For example, for Arbiter 1, each occurrence of gr_2 in an event sequence w is preceded by an equal number of occurrences of gr_1 and rel_1:

$$|w|_{\mathrm{gr}_1\,\mathrm{gr}_2} = |w|_{\mathrm{rel}_1\,gr_2} \text{ and, symmetrically, } |w|_{\mathrm{gr}_2\,\mathrm{gr}_1} = |w|_{\mathrm{rel}_2\,\mathrm{gr}_1}.$$

(2) Whenever a grant is given to a child, the number of grants received from the parent exceeds the number of releases sent to it by exactly 1. For example, for

Arbiter 1, each occurrence of gr_1 or gr_2 is preceded by one more occurrence of gr_0 than of rel_0:

$$|w|_{\mathrm{gr}_0\,\mathrm{gr}_i} = |w|_{\mathrm{rel}_0\,\mathrm{gr}_i} + |w|_{\mathrm{gr}_i}, \text{ for } i = 1, 2.$$

(3) Whenever a release is sent to, or a grant received from, the parent, the number of releases received from each child equals the number of grants given to that child. For Arbiter 1:

$$|w|_{\mathrm{gr}_i\,\mathrm{gr}_0} = |w|_{\mathrm{rel}_i\,\mathrm{gr}_0} \text{ and } |w|_{\mathrm{gr}_i\,\mathrm{rel}_0} = |w|_{\mathrm{rel}_i\,\mathrm{rel}_0}, \text{ for } i = 1, 2.$$

(4) The differences between the corresponding numbers of grants and releases only ever take values in $\{0, 1\}$. For Arbiter 1:

$$|w|_{\mathrm{gr}_i\,\mathrm{rel}_i} + |w|_{\mathrm{rel}_i\,\mathrm{gr}_i} = |w|_{\mathrm{gr}_i\,\mathrm{gr}_i} + |w|_{\mathrm{rel}_i\,\mathrm{rel}_i} + |w|_{\mathrm{rel}_i}, \text{ for } i = 0, 1, 2.$$

Combined, the subsequence invariants (1) - (4) of all arbiter nodes imply that the arbiter tree guarantees mutual exclusion among its clients.

In this paper, we present algorithms for the synthesis of subsequence invariants that automatically compute all invariants of an automaton for a given set of subsequences. Since the set of invariants is in general not finite, it is represented algebraically by a finite set of generators. Based on the synthesis algorithms, we propose the following verification technique for subsequence invariants:

To prove a desired system property φ, we first choose, for each process, a set U of relevant subsequences and then synthesize a basis of the subsequence invariants over U. The invariants computed for each individual process translate to invariants of the system. If φ is a linear combination of the system invariants, we know that φ itself is a valid invariant.

The only manual step in this technique is the choice of an appropriate set of subsequences, which depends on the complexity of the interaction between the processes. A practical approach is therefore to begin with a small set of subsequences and then incrementally compute a growing set of invariants based on a growing set of subsequences until φ is proved.

In the following, due to space constraints, all proofs have been omitted. We refer the reader to the full version of this paper [6].

Related work. There is a significant body of work on the generation of invariants over program variables, ranging from heuristics (cf. [7]), to methods based on abstract interpretation (cf. [1, 2, 4, 14]) and constraint solving (cf. [3]). The key difference to our approach is that, while these approaches aim at finding a concise characterization of a complex state space, we aim at finding a concise representation of a complex process interaction. T-invariants, which relate the number of firings of different transitions in a Petri net, have a similar motivation (cf. [11]), but are not applied in the compositional manner of subsequence invariants.

Subsequence occurrences have, to the best of our knowledge, not been used in verification before. However, there has been substantial interest in subsequences in the context of formal languages, in particular in connection with Parikh matrices and their generalizations; see, for example, [5, 8, 10, 13], as well as Parikh's original paper [12], introducing Parikh images.

Subsequences are also used in machine learning, in the context of kernel-based methods for text classification [9]; here the focus is on their use as characteristic values of given pieces of text, not on the characterization of languages or systems by constraints on their possible values.

2 Preliminaries

Linear algebra. For a given finite set U, the *real vector space* $\mathbb{R}^U$ *generated by* U consists of all tuples $\phi = (\phi_u)_{u \in U}$ of real numbers indexed by the elements of U. For a given set of vectors $\phi^1, \ldots, \phi^k \in \mathbb{R}^U$, the subspace $\mathrm{span}(\phi^1, \ldots, \phi^k)$ *spanned by* $\phi^1, \ldots, \phi^k$ consists of all linear combinations $\lambda_1 \phi^1 + \ldots \lambda_k \phi^k$ for $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$. We assume that the set U is equipped with a total ordering $<$, i.e., $U = \{u^1, \ldots, u^m\}$ with $u^1 < \cdots < u^m$. We write vectors as tuples $(\phi_{u^1}, \ldots, \phi_{u^m})$ according to this order. The *pivot element* $pivot(\phi)$ is the $<$-least element u such that ϕ_u is nonzero.

A set B of linearly independent vectors is a *basis* for a subspace $H \subseteq \mathbb{R}^U$ iff $H = \mathrm{span}(B)$. When we collect the basis vectors of a subspace, we ensure the linear independence of the vectors with the following construction: To add a new vector η to a set $\{\phi^1, \ldots, \phi^k\}$ of vectors, we consider, for each vector ϕ^i, the pivot element $u^i := pivot(\phi^i)$ and reduce η to $\eta - (\eta_{u^i}/\phi^i_{u^i})\phi^i$. For the resulting vector η' we know that $\eta'_{u^i} = 0$ for all i. If $\eta' = \mathbf{0}$, then the new set of vectors is the same as the original set $\{\phi^1, \ldots, \phi^k\}$; otherwise, we reduce each vector ϕ^i from the original set to $\psi^i := \phi^i - (\phi^i_{u^i}/\eta_{u^i})\eta$, resulting in the new set $\{\psi^1, \ldots, \psi^k, \eta'\}$.

As an example, consider the set of vectors $\{\phi^1, \phi^2\} \subset \mathbb{R}^{\{1,\ldots,5\}}$, where $\phi^1 = (1, 2, 0, -1, 1)^T$ and $\phi^2 = (0, 0, 1, 2, -2)^T$, with pivot elements 1 and 3, respectively. A new vector $\eta = (1, 1, 2, 2, -1)^T$ would first be reduced to $(0, -1, 2, 3, -2)^T$ (by subtracting ϕ^1), and then to $\eta' = (0, -1, 0, -1, 2)^T$ (by subtracting $2\phi^2$). Reducing ϕ^1, we obtain $\psi^1 = (1, 0, 0, -3, 5)^T$, resulting in the new set $\{(1, 0, 0, -3, 5)^T, (0, 0, 1, 2, -2)^T, (0, -1, 0, -1, 2)^T\}$.

The *orthogonal complement* $H^\perp$ of a subspace $H \subseteq \mathbb{R}^U$ consists of the vectors that are orthogonal to those in H, i.e., all vectors ψ where the scalar product $\psi \cdot \phi = \sum_{u \in U} \psi_u \phi_u$ is zero for all $\phi \in H$. Given a basis B for H that has been reduced as described above, a basis for $H^\perp$ is obtained as follows:

Let $V \subseteq U$ be the set of all $u \in U$ which are not the pivot element of any $\phi \in B$. For each $u \in V$, define a vector ψ^u by $\psi^u_u = 1, \psi^u_v = 0$ for all $v \in V \setminus \{u\}$, and for each $\phi \in B$, $\psi^u_{pivot(\phi)} = -\phi_u/\phi_{pivot(\phi)}$. For example, given the basis $B = \{(1, 0, 0, -3, 5)^T, (0, 0, 1, 2, -2)^T, (0, -1, 0, -1, 2)^T\}$, we have that $V = \{4, 5\}$, and therefore obtain the basis vectors $\psi^4 = (3, -1, -2, 1, 0)^T$ and $\psi^5 = (-5, 2, 2, 0, 1)^T$ for $\mathrm{span}(B)^\perp$.

Alphabets and Sequences. An alphabet is a finite set of symbols. For an alphabet A, A^* is the set of finite sequences over A. The empty sequence is denoted by ϵ, the composition of two sequences $v, w \in \mathrm{A}^*$ by $v.w$, and the length of a sequence w by $|w|$.

For alphabets $\mathrm{A}_1 \subseteq \mathrm{A}_2$, the *projection* $w \downarrow_{\mathrm{A}_1}$ of a sequence $w \in \mathrm{A}_2^*$ onto A_1 is defined recursively by

$$\epsilon \downarrow_{\mathrm{A}_1} = \epsilon,\ (w.a) \downarrow_{\mathrm{A}_1} = \begin{cases} (w \downarrow_{\mathrm{A}_1}).a & \text{if } a \in \mathrm{A}_1, \\ w \downarrow_{\mathrm{A}_1} & \text{otherwise.} \end{cases}$$

We equip A with a total order $<$, and A^* with the corresponding length-lexicographical ordering given by $u <_{llex} v$ iff either

- $|u| < |v|$ or
- $|u| = |v|$, and there are $x, y, z \in \mathrm{A}^*, a, b \in \mathrm{A}$ with $a < b, u = xay, v = xbz$.

In particular, elements ϕ of the vector space $\mathbb{R}^U$, generated by a finite subset $U \subset \mathrm{A}^*$, are written according to this order, i.e., $\phi = (\phi_{u^1}, \ldots, \phi_{u^n})$ for $U = \{u^1, \ldots, u^n\}, u^1 <_{llex} \cdots <_{llex} u^n$.

Communicating automata. We consider concurrent systems that are given as a set of communicating finite-state automata. A (nondeterministic) *finite automaton* $P = (Q_P, \mathrm{A}_P, q_P^0, T_P)$ consists of

- a finite set Q_P of locations,
- a finite alphabet A_P of synchronization events,
- an initial location $q_P^0 \in Q_P$, and
- a transition relation $T_P \subseteq Q_P \times \mathrm{A}_P \times Q_P$.

When dealing with automata $P_1, \ldots, P_n$, we use i as the subscript instead of P_i.

We denote $(q, a, r) \in T_P$ by $q \xrightarrow{a}_P r$. For a sequence $w = w_1 \ldots w_n \in \mathrm{A}_P^*$, $q \xrightarrow{w}_P r$ iff $q \xrightarrow{w_1}_P \cdots \xrightarrow{w_n}_P r$. The *language* of a location $q \in Q_P$ is the set $\mathrm{L}(q) := \{w \in \mathrm{A}_P^* : q^0 \xrightarrow{w}_P q\}$; q is *reachable* iff $\mathrm{L}(q) \neq \emptyset$. We assume in the following that our automata only contain reachable locations. For a subset $Q' \subseteq Q_P$ of the locations, the language of Q' is the union of all languages of the locations in Q'. The language of an automaton P is the language of its locations, $\mathrm{L}(P) := \mathrm{L}(Q_P)$.

A set $\{P_1, \ldots, P_n\}$ of finite automata defines a system automaton $S = (Q, \mathrm{A}, q^0, \rightarrow)$, where $Q = Q_1 \times \cdots \times Q_n$, $\mathrm{A} = \mathrm{A}_1 \cup \cdots \cup \mathrm{A}_n$, and $(q_1, \ldots, q_n) \xrightarrow{a} (r_1, \ldots, r_n)$ iff for all $i \in \{1, \ldots, n\}$ either

- $a \in \mathrm{A}_i$ and $q_i \xrightarrow{a}_i r_i$, or
- $a \notin \mathrm{A}_i$ and $q_i = r_i$.

The language $\mathrm{L}(S)$ of S thus consists of all sequences w over A, such that, for each automaton P_i, the projection $w{\downarrow_{\mathrm{A}_i}}$ to the alphabet A_i is in the language $\mathrm{L}(P_i)$.

3 Subsequence Invariants

Let $P = (Q, \mathrm{A}, q^0, T)$ be a finite automaton. We define the subsequence invariants of P relative to a given finite, prefix-closed set of sequences $U = \{u^1, \ldots, u^n\} \subset \mathrm{A}^*$, which we call the *set of subsequences*.

Given two sequences $u = u_1 \dots u_k$ and $w = w_1 \dots w_n \in \mathrm{A}^*$, the *set of occurrences of u as a subsequence* in w is $[w]_u := \{(i_1, \dots, i_k) : 1 \le i_1 < \dots < i_k \le n, w_{i_j} = u_j \text{ for all } j\}$. For example, $[aababb]_{ab} = \{(1,3), (1,5), (1,6), (2,3), (2,5), (2,6), (4,5), (4,6)\}$. The sizes of these sets define the *numbers of occurrences* $|w|_u := \|[w]_u\|$. These numbers can be computed recursively, using the recurrence [8]

$$|w|_\epsilon = 1, \qquad |\epsilon|_{u.b} = 0, \qquad |w.a|_{u.b} = \begin{cases} |w|_{u.b} + |w|_u & \text{if } a = b, \\ |w|_{u.b} & \text{otherwise,} \end{cases}$$

for all $u, w \in \mathrm{A}^*, a, b \in \mathrm{A}$. This gives rise to a mapping $|.|_U$ from A^* into $\mathbb{R}^U$ defined by $|w|_U = (|w|_{u^1}, \dots, |w|_{u^n})$.

For any subset $Q' \subseteq Q$, the *subsequence hull* of Q' is the subspace $H_{Q'}$ of $\mathbb{R}^U$ spanned by the subsequence occurrences $\{|w|_U : w \in \mathrm{L}(Q')\}$.

Definition 1. *A* subsequence invariant *for $Q' \subseteq Q$ over U is a vector $\phi \in \mathbb{R}^U$ such that for all $w \in \mathrm{L}(Q')$, $\sum_{u \in U} \phi_u |w|_u = 0$.*

The subsequence invariants for Q' define a linear subspace $I_{Q'} \subseteq \mathbb{R}^U$, which is the orthogonal complement of $H_{Q'}$ in $\mathbb{R}^U$. Special cases are the *local subsequence invariants* $I_q = I_{\{q\}}$ at $q \in Q$ and the *global invariants* of P, $I_P = I_Q$. The spaces of the invariants satisfy the relation $I_{Q'} = \bigcap_{q \in Q'} I_q$.

The sequences that satisfy a given set of subsequence invariants form a context-sensitive language [13]. The expressiveness of subsequence invariants is, however, incomparable to the regular and context-free languages. For example, subsequence invariants can characterize the context-sensitive language $\{a^n b^n c^n : n \in \mathbb{N}\} = \{w \in \mathrm{A}^* : |w|_a = |w|_b, |w|_b = |w|_c, |w|_{ba} = 0, |w|_{cb} = 0\}$, but not the regular language $\{a.w : w \in \mathrm{A}^*\}$ for some $a \in \mathrm{A}$.

Requiring invariants to be linear equalities may appear restrictive. In the remainder of this section we illustrate the expressive power of subsequence invariants by translating two useful types of invariants, *conditional* and *disjunctive* invariants, to equivalent linear subsequence invariants.

Resolving conditions. Properties (1)–(3) of the arbiter tree discussed in the introduction are examples of *conditional invariants*, stating that a linear equality over the numbers $|w|_u$ should hold whenever some event $a \in \mathrm{A}$ occurs. Obviously, the equality must be in I_{E_a}, where E_a is the set of locations in which an a-transition can occur. We can resolve the event condition to obtain a global statement, using subsequences no more than one symbol longer than those in U, as follows:

Theorem 1. *Let $a \in \mathrm{A}$ and $E_a := \{q \in Q : q \xrightarrow{a} r \text{ for some } r \in Q\}$. Then $\sum_{u \in U} \phi_u |w|_u = 0$ for all $w \in \mathrm{L}(E_a)$ if and only if $\sum_{u \in U} \phi_u |w|_{u.a} = 0$ for all $w \in \mathrm{L}(P)$.*

Thus, for example, the condition that the number of releases received from the left child must equal the number of grants given to it whenever a grant is given to the right child, i.e., $|w|_{\mathrm{gr}_1} = |w|_{\mathrm{rel}_1}$ for all $w \in E_{\mathrm{gr}_2}$, is equivalent to $|w|_{gr_1.gr_2} = |w|_{rel_1.gr_2}$ for all $w \in \mathrm{L}(P)$.

Resolving disjunctions. Consider now the fourth statement in the introductory example: The differences between the corresponding numbers of grants and releases only ever take values in $\{0,1\}$. Such a *disjunctive condition* can be translated in two steps into an equivalent linear equation: The condition is first transformed into a polynomial equation (Step 1), and then reduced, using algebraic dependencies, to an equivalent linear equation (Step 2).

Step 1 is simple: The condition $\sum_{u\in U}\phi_u|w|_u \in \{c_1,\dots,c_k\}$ is equivalent to $(\sum_{u\in U}\phi_u|w|_u - c_1)\cdots(\sum_{u\in U}\phi_u|w|_u - c_k) = 0$.

For the transformation of the resulting polynomial equation into a linear equation, we define, as an auxiliary notion, the set of *coverings* of $x \in \mathrm{A}^*$ by u and v to be

$$[x]_{u,v} := \{((i_1,\dots,i_k),(j_1,\dots,j_m)) : i_1 < \dots < i_k, j_1 < \dots < j_m, \\ u = x_{i_1}\dots x_{i_k}, v = x_{j_l}\dots x_{j_m}, \\ \{i_1,\dots,i_k,j_1,\dots,j_m\} = \{1,\dots,|x|\}\},$$

i.e., the set of pairs of occurrences of u and v as subsequences of x such that every index in $1,\dots,|x|$ is used in at least one of them. For example,

$$[aabaa]_{aaa,aba} = \{((1,2,4),(1,3,5)),((1,2,4),(2,3,5)),((1,2,5),(1,3,4)),((1,2,5),(2,3,4)), \\ ((1,4,5),(2,3,4)),((1,4,5),(2,3,5)),((2,4,5),(1,3,4)),((2,4,5),(1,3,5))\}.$$

Let $|w|_{u,v} = \|[w]_{u,v}\|$ denote the number of coverings, which can be computed recursively as follows:

$$|w|_{u,\epsilon} = |w|_{\epsilon,u} = \begin{cases} 1 & \text{if } u = w, \\ 0 & \text{otherwise,} \end{cases} \qquad |\epsilon|_{u,v} = \begin{cases} 1 & \text{if } u = v = \epsilon, \\ 0 & \text{otherwise,} \end{cases}$$

$$|w.a|_{u.b,v.c} = \begin{cases} |w|_{u,v} + |w|_{u.b,v} + |w|_{u,v.c} & \text{if } b = a = c, \\ |w|_{u,v.c} & \text{if } b = a \neq c, \\ |w|_{u.b,v} & \text{if } b \neq a = c, \\ 0 & \text{if } b \neq a \neq c. \end{cases}$$

It is easy to see that for every $u,v \in \mathrm{A}^*$, the set $C(u,v) := \{x \in \mathrm{A}^* : [x]_{u,v} \neq \emptyset\}$ of sequences coverable by u and v is finite, since it cannot contain sequences longer than $|u| + |v|$.

Theorem 2 *(See Theorem 6.3.18 in [8] for an equivalent statement to (2))*

1. *For all $u,v,w \in \mathrm{A}^*$, there is a bijection between $[w]_u \times [w]_v$ and $\biguplus_{x\in C(u,v)}([x]_{u,v} \times [w]_x)$, and therefore,*
2. *For all $u,v,w \in \mathrm{A}^*$, $|w|_u|w|_v = \sum_{x\in C(u,v)}|x|_{u,v}|w|_x$.*

Simple examples for Theorem 2 are the equalities $|w|_a^2 = 2|w|_{aa} + |w|_a$ and $|w|_a|w|_b = |w|_{ab} + |w|_{ba}$. For $u = ab$ and $v = ba$, we obtain the equality $|w|_{ab}|w|_{ba} = |w|_{aba} + |w|_{bab} + |w|_{abab} + 2|w|_{abba} + 2|w|_{baab} + |w|_{baba}$.

The degree k polynomial equation $p(|w|_{u^1}, \ldots, |w|_{u^n}) = 0$ resulting from Step 1 can then be transformed into a linear equation using the equalities from Theorem 2. This linear equation involves subsequences of length up to kl, where l is the maximum length of any $u \in U$.

Example: For property (4) from the introduction, we obtain

$$\begin{aligned}
&|w|_{\mathrm{gr}_i} - |w|_{\mathrm{rel}_i} \in \{0, 1\} \\
&\quad \Leftrightarrow (|w|_{\mathrm{gr}_i} - |w|_{\mathrm{rel}_i})(|w|_{\mathrm{gr}_i} - |w|_{\mathrm{rel}_i} - 1) = 0 \\
&\quad \Leftrightarrow |w|^2_{\mathrm{gr}_i} - 2|w|_{\mathrm{gr}_i}|w|_{\mathrm{rel}_i} + |w|^2_{\mathrm{rel}_i} - |w|_{\mathrm{gr}_i} + |w|_{\mathrm{rel}_i} = 0 \\
&\quad \Leftrightarrow |w|_{\mathrm{gr}_i \,.\, \mathrm{gr}_i} + |w|_{rel_i.rel_i} + |w|_{rel_i} = |w|_{\mathrm{gr}_i \,.\, \mathrm{rel}_i} + |w|_{\mathrm{rel}_i \,.\, \mathrm{gr}_i}.
\end{aligned}$$

This technique can also handle more complicated constraints: An alternative characterization of Arbiter 1 is given by the requirement that for all $w \in \mathrm{L}(P)$,

$$\begin{pmatrix} |w|_{\mathrm{gr}_0} - |w|_{\mathrm{rel}_0} \\ |w|_{\mathrm{gr}_1} - |w|_{\mathrm{rel}_1} \\ |w|_{\mathrm{gr}_2} - |w|_{\mathrm{rel}_2} \end{pmatrix} \in \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \right\}.$$

Note that the possible values for the linear expressions are mutually dependent. The set of vectors on the right-hand side can be characterized as the set of all $(x, y, z)^T$ for which $x^2 - x, y^2 - y, z^2 - z, xy - y, xz - z$ and yz are all zero. Using Theorem 2, we again obtain a set of linear subsequence constraints. In general, we have:

Theorem 3. *Let* $\|U\| = n, \max\{|u| : u \in U\} = l, M \in \mathbb{R}^{k \times n}$, *and* $\phi^1, \ldots, \phi^m \in \mathbb{R}^k$. *Then the constraint given by* $M|w|_U \in \{\phi^1, \ldots, \phi^m\}$ *is equivalent to a finite set of linear subsequence constraints involving subsequences of length* $\leq ml$.

4 Computing Subsequence Invariants

In this section, we present two algorithms for computing the subsequence invariants of a given finite automaton $P = (Q, \mathrm{A}, q^0, T)$ with respect to a finite, prefix-closed set $U \subset \mathrm{A}^*$ of subsequences. The first algorithm is generally applicable. The second algorithm is a more efficient solution that is applicable if the state graph is strongly connected.

4.1 The General Algorithm

The subsequence invariants are computed using matrices F_a representing the effect of appending $a \in \mathrm{A}$, which are defined by

$$F_a = (f_{u,v})_{u,v \in U} : f_{u,v} = \begin{cases} 1 & \text{if } u \in \{v, v.a\}, \\ 0 & \text{otherwise.} \end{cases}$$

For example, for $U = \{\epsilon, a, b, aa, ab, ba, bb\}$,

$$|w.a|_U = \begin{pmatrix} 1&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0 \\ 0&1&0&1&0&0&0 \\ 0&0&0&0&1&0&0 \\ 0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&1 \end{pmatrix} |w|_U, |w.b|_U = \begin{pmatrix} 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0 \\ 1&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0 \\ 0&1&0&0&1&0&0 \\ 0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&1 \end{pmatrix} |w|_U.$$

Data: Automaton $P = (Q, \mathrm{A}, q^0, T)$, finite prefix-closed $U \subset \mathrm{A}^*$
Result: Bases B_q for the subspaces $H_q = \mathrm{span}(|w|_U : w \in \mathrm{L}(q))$
// Initialization:
foreach $q \in Q$ **do** $B_q := \emptyset$;
// B_{q^0} initially contains $\{|\epsilon|_U\}$
$B_{q^0} := \{(1, 0, \ldots, 0)^T\}$;
// The open list, containing pairs (q, ϕ) to be explored
$O := \{(q^0, (1, 0, \ldots, 0)^T)\}$;
// Basis construction:
while $O \neq \emptyset$ **do**
  take (q, ϕ) from O;
  foreach $q \xrightarrow{a} r$ **do**
    $\psi := F_a \phi$;
    begin reduce ψ with B_r:
      foreach $\eta \in B_r$ **do**
        $v := \min\{u \in U : \eta_u \neq 0\}$;
        $\psi := \psi - (\psi_v / \eta_v)\eta$;
    end
    if $\psi \neq \mathbf{0}$ **then**
      $B_r := B_r \cup \{\psi\}$;
      $O := O \cup \{(r, \psi)\}$;

Fig. 2. Fixpoint iteration computing the subspaces H_q

These matrices are easily seen to be unit lower triangular matrices (recall that U is ordered by $<_{llex}$) and thus have determinant 1; their inverses are

$$F_a^{-1} = (b_{u,v})_{u,v \in U} : b_{u,v} = \begin{cases} (-1)^k & \text{if } u = v.a^k, k \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

To compute the invariants, we determine, for all $q \in Q$, a basis of the subspace $H_q = \mathrm{span}(\{|w|_U : w \in \mathrm{L}(q)\})$, using the fixpoint iteration shown in Figure 2.

Theorem 4. *1. The sets B_q computed by the fixpoint iteration shown in Figure 2 are bases for the vector spaces H_q spanned by $\{|w|_U : w \in \mathrm{L}(q)\}$.*
2. When called for an automaton $P = (Q, \mathrm{A}, q^0, T)$ with $\|T\| = m$ and $U \subset \mathrm{A}^*$ with $\|U\| = n$, the fixpoint iteration terminates in time $O(mn^3)$.

4.2 An Optimized Algorithm for Strongly-Connected Automata

If P is strongly connected, i.e., there is a path from q to r for all locations $q, r \in Q$, we can improve the construction of the invariants. For $w = w_1 \ldots w_n$ such that $q \xrightarrow{w} r$, the composition $F_w = F_{w_n} \ldots F_{w_1}$ is an isomorphism from H_q to its image $F_w(H_q) \subseteq H_r$, implying in particular $\dim(H_q) \leq \dim(H_r)$. In the strongly connected case, this implies $\dim(H_q) = \dim(H_r)$ for all q, r, and $H_r = F_w(H_q)$, i.e., F_w is an isomorphism from H_q to H_r when $q \xrightarrow{w} r$.

Data: Automaton $P = (Q, \mathrm{A}, q^0, T)$, finite prefix-closed $U \subset \mathrm{A}^*$
Result: Bases B_q for the subspaces $H_q = \mathrm{span}(|w|_U : w \in \mathrm{L}(q))$
// Initialization:
$M_{q^0} := IC := \emptyset$;
$O := \{q^0\}$;
// Exploration:
while $O \neq \emptyset$ **do**
  take q from O;
  foreach $q \xrightarrow{a} r$ **do**
    $N := F_a M_q$;
    if M_r *not yet defined* **then**
      define $M_r := N$;
      $O := O \cup \{r\}$;
    else if $M_r \neq N$ **then**
      $C := C \cup \{M_r^{-1} N\}$;
// Basis construction:
$B_{q^0} := \{(1, 0, \ldots, 0)\}$;
$O := \{(1, 0, \ldots, 0)\}$;
while $O \neq \emptyset$ **do**
  take ϕ from O; **foreach** $M \in C$ **do**
    $\psi := M\phi$;
    begin reduce ψ with B_{q^0}:
      foreach $\eta \in B_{q^0}$ **do**
        $v := \min\{u \in U : \eta_u \neq 0\}$;
        $\psi := \psi - (\psi_v / \eta_v)\eta$;
    end
    if $\psi \neq \mathbf{0}$ **then**
      $B_{q^0} := B_{q^0} \cup \{\psi\}$;
      $O := O \cup \{\psi\}$;
foreach $q \in Q \setminus \{q^0\}$ **do**
  $B_q := \{M_q \phi : \phi \in B_{q^0}\}$

Fig. 3. Local fixpoint iteration computing the subspaces H_q

The local fixpoint iteration shown in Figure 3 exploits this observation by finding isomorphisms $M_q : H_{q^0} \to H_q$ for all $q \in Q$ as well as a set C of automorphisms of H_{q^0} corresponding to a cycle basis of the automaton. The matrices $C_i \in C$ are then used to compute a basis of H_{q^0}. For all other $q \in Q$, H_q is obtained via M_q. The main advantage of this algorithm is the lower number of reduction steps if the cycle degree $\|T\| - \|Q\| + 1$ of P is small:

Theorem 5. *1. The set B_{q^0} computed by the local fixpoint iteration shown in Figure 3 forms a basis of H_{q^0}.*

2. When called for an automaton $P = (Q, \mathrm{A}, q^0, T)$ with $\|T\| = m$ and cycle degree $\gamma := \|T\| - \|Q\| + 1$, and $U \subset \mathrm{A}^*$ with $\|U\| = n$, the local fixpoint iteration terminates in time $O(mn^2 + \gamma n^3)$.

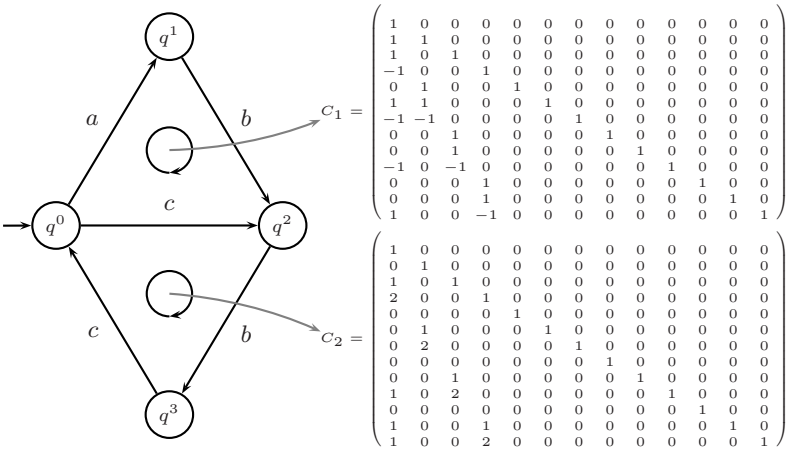


Fig. 4. Example for the local fixpoint construction

Example: Consider the automaton in Figure 4. Using $U = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc\}$, we compute B_{q^0} as follows:

1. Initialization: $M_{q^0} = I, C = \emptyset, B = \{\phi^0\}$, where $\phi^0 = (1, 0, \ldots, 0)$;
2. Exploration:
$$q^0 \xrightarrow{a} q^1 : M_{q^1} = F_a;$$
$$q^0 \xrightarrow{c} q^2 : M_{q^2} = F_c;$$
$$q^1 \xrightarrow{b} q^2 : \text{add } C_1 = F_c^{-1} F_b F_a \text{ to } C;$$
$$q^2 \xrightarrow{b} q^3 : M_{q^3} = F_b F_c;$$
$$q^3 \xrightarrow{c} q^0 : \text{add } C_2 = F_c F_b F_c \text{ to } C;$$
C_1 and C_2 correspond to the two basic undirected cycles $q^0 \xrightarrow{a} q^1 \xrightarrow{b} q^2 \xleftarrow{c} q^0$ and $q^0 \xrightarrow{c} q^2 \xrightarrow{b} q^3 \xrightarrow{c} q^0$.
3. Basis construction: Successively extending B by reducing and adding $L_1\phi^0$, $L_2\phi^0$, $L_1^2\phi^0$, $L_2L_1\phi^0$, $L_1L_2\phi^0$, $L_2^2\phi^0$, we obtain basis vectors

$$\begin{aligned}
\phi^0 &= (1, 0, 0, \ 0, \ 0, 0, \ 0, \ 0, 0, \ 0, \ 0, \ 0, \ 0)^T,\\
\phi^1 &= (0, 1, 0, -3, 0, 0, -3, 0, 0, -2, \ 0, \ 2, \ 6)^T,\\
\phi^2 &= (0, 0, 1, \ 2, \ 0, 0, \ 0, \ 0, 0, \ 1, \ 0, \ 1, \ 1)^T,\\
\phi^3 &= (0, 0, 0, \ 0, \ 1, 0, -3, 0, 0, \ 0, \ -3, \ 0, \ 9)^T,\\
\phi^4 &= (0, 0, 0, \ 0, \ 0, 1, \ 2, \ 0, 0, \ 0, \ 0, \ -3, \ -6)^T,\\
\phi^5 &= (0, 0, 0, \ 0, \ 0, 0, \ 0, \ 1, 0, -3, \ 2, \ 0, \ -6)^T,\\
\phi^6 &= (0, 0, 0, \ 0, \ 0, 0, \ 0, \ 0, 1, \ 2, \ 0, \ 2, \ 4)^T.
\end{aligned}$$

All further products $L_i\phi^j$ reduce to $\mathbf{0}$.

4. Local invariant generation: Computing the orthogonal complement, we obtain the following basis for I_{q^0}:

$$\begin{aligned}
\psi^1 &= (0,\ 3,\ -2,1,\ 0,\ \ 0,\ 0,\ 0,\ \ 0,\ 0,0,0,0)^T,\\
\psi^2 &= (0,\ 3,\ \ 0,\ 0,\ 3,\ -2,1,\ 0,\ \ 0,\ 0,0,0,0)^T,\\
\psi^3 &= (0,\ 2,\ -1,0,\ 0,\ \ 0,\ 0,\ 3,\ -2,1,0,0,0)^T,\\
\psi^4 &= (0,\ 0,\ \ 0,\ 0,\ 3,\ \ 0,\ 0,-2,\ 0,\ 0,1,0,0)^T,\\
\psi^5 &= (0,-2,-1,0,\ 0,\ \ 3,\ 0,\ 0,\ -2,0,0,1,0)^T,\\
\psi^6 &= (0,-6,-1,0,-9,\ 6,\ 0,\ 6,\ -4,0,0,0,1)^T.
\end{aligned}$$

   For example, ψ^1 represents the invariant $3|w|_a - 2|w|_b + |w|_c = 0$, ψ^2 the invariant $3|w|_a + 3|w|_{aa} - 2|w|_{ab} + |w|_{ac} = 0$.
5. Global invariant generation: Adding the vectors $M_q\phi^i$ to B and computing the orthogonal complement, we obtain the single global invariant $3|w|_{aa} - 2|w|_{ba} + |w|_{ca} = 0$.

5 From Process to System Invariants

A key advantage of subsequence invariants is that invariants that have been computed for an individual automaton are immediately inherited by the full system and can therefore by composed by simple conjunction.

Theorem 6. *Let $S = \{P_1, \ldots, P_n\}$ be a system of communicating finite automata. If $\sum_{u\in U} \phi_u|w|_u = 0$ is a global subsequence invariant for P_i over $U \subset \mathrm{A}_i^*$, then $\sum_{u\in U} \phi_u|w|_u = 0$ also holds for all $w \in \mathrm{L}(S)$.*

The system S may satisfy additional invariants, not covered by Theorem 6, that refer to interleavings of sequences from A_i^* with sequences from a different A_j^*. In the following, we present two methods for obtaining such additional invariants.

5.1 System Invariants Obtained by Projection

The first approach works similarly to the resolution of conditions in the Section 3. It uses the fact that given any subsequence invariant for S, we can obtain a new subsequence invariant by appending the same symbol to all involved subsequences:

Theorem 7. *Let $\sum_{u\in U} \phi_u|w|_u = 0$ for all $w \in \mathrm{L}(S)$, and $a \in \mathrm{A}$. Then we also have $\sum_{u\in U} \phi_u|w|_{u.a} = 0$ for all $w \in \mathrm{L}(S)$.*

Example: Consider a system containing the automaton from Figure 4. From the invariant $(3|w|_{aa} - 2|w|_{ba} + |w|_{ca})|w|_{ad} = 0$ we obtain the new invariants $3|w|_{aad} - 2|w|_{bad} + |w|_{cad} = 0$, $3|w|_{aaad} - 2|w|_{baad} + |w|_{caad} = 0$, and $3|w|_{aada} - 2|w|_{bada} + |w|_{cada} = 0$ by appending d, ad, and da, respectively.

5.2 System Invariants Obtained by Algebraic Dependencies

The equalities in Theorem 2 can be used to derive new invariants from a given set of subsequence invariants:

Let $\sum_{u\in U}\phi_u|w|_u = 0$ for all $w \in \mathrm{L}(S)$, and $v \in \mathrm{A}^*$. Then obviously, $\sum_{u\in U}\phi_u|w|_u|w|_v$ is also zero; Using the equalities $|w|_u|w|_v = \sum_{x\in C(u,v)}|x|_{u,v}|w|_x$, this can be transformed into new linear subsequence invariants $\sum_{u\in U}\sum_{x\in C(u,v)}\phi_u|x|_{u,v}|w|_x = 0$.

Example: Consider a system containing the automaton from Figure 4. It contributes the invariant $3|w|_{aa} - 2|w|_{ba} + |w|_{ca} = 0$ for all $w \in \mathrm{L}(S)$. For $v = ad$, Theorem 2 provides the algebraic dependencies

$$
\begin{aligned}
|w|_{aa}|w|_{ad} &= 2|w|_{aad} + |w|_{ada} + 3|w|_{aaad} + 2|w|_{aada} + |w|_{adaa},\\
|w|_{ba}|w|_{ad} &= |w|_{bad} + |w|_{abad} + |w|_{abda} + |w|_{adba} + 2|w|_{baad} + |w|_{bada},\\
|w|_{ca}|w|_{ad} &= |w|_{cad} + |w|_{acad} + |w|_{acda} + |w|_{adca} + 2|w|_{caad} + |w|_{cada},
\end{aligned}
$$

which can be used to obtain from $(3|w|_{aa} - 2|w|_{ba} + |w|_{ca})|w|_{ad} = 0$ the new subsequence invariant $6|w|_{aad}+3|w|_{ada}+9|w|_{aaad}+6|w|_{aada}+3|w|_{adaa}-2|w|_{bad}-2|w|_{abad}-2|w|_{abda}-2|w|_{adba}-4|w|_{baad}-2|w|_{bada}+|w|_{cad}+|w|_{acad}+|w|_{acda}+|w|_{adca}+2|w|_{caad}+|w|_{cada} = 0$.

Using the invariants from the previous example, the new invariant reduces to $3|w|_{aad}+3|w|_{ada}+3|w|_{aaad}+3|w|_{aada}+3|w|_{adaa}-2|w|_{abad}-2|w|_{abda}-2|w|_{adba}+|w|_{acad}+|w|_{acda}+|w|_{adca} = 0$.

6 Incremental Invariant Generation

For the invariant generation algorithms of Section 4, we considered the set U of subsequences as given and fixed. In practice, however, the set of subsequences depends on the complexity of the interaction between the processes, and is therefore not necessarily known in advance. In this section, we present an incremental method that allows for growing sets of subsequences.

Let $P = (Q_P, \mathrm{A}_P, q_P^0, T_P)$ be an automaton and $U \subset \mathrm{A}^*$ be finite and prefix-closed. Let $V = U \uplus \{v\}$ again be prefix-closed, i.e. $v = u.a$ for some $u \in U, a \in \mathrm{A}$.

Theorem 8. *Assume that for $q \in Q_P$ and the set of subsequences U, a basis of the space $H_{q,U} = \mathrm{span}(|w|_U : w \in \mathrm{L}(q))$ has already been computed, consisting of the vectors $\phi^1, \ldots, \phi^k$. Then either*

1. *$H_{q,V}$ is spanned by vectors $\psi^1, \ldots, \psi^k$ such that $\psi^j_u = \phi^j_u$ for all $u \in U$, or*
2. *$H_{q,V}$ is spanned by the vectors $\psi^1, \ldots, \psi^k, \eta$ given by:*
   - *$\psi^j_u = \phi^j_u$ for all $u \in U$, and $\psi^j_v = 0$;*
   - *$\eta_u = 0$ for all $u \in U$, and $\eta_v = 1$.*

All invariants obtained for U remain valid; in the first case, we additionally obtain a new invariant $|w|_v - \sum_{i=1}^{k}(\psi^i_v/\psi^i_{u^i})|w|_{u^i} = 0$, where $u^i = pivot(\psi^i)$ for all i, while in the second case, the set of invariants is unchanged.

Example: Consider again the automaton in Figure 4. Starting with the smaller set of subsequences $U = \{\epsilon, a, b, c\}$, we obtain the basis $\{(1,0,0,0)^T, (0,1,0,-3)^T, (0,0,1,2)^T\}$ for $H_{q^0,U}$, along with the single local invariant $3|w|_a - 2|w|_b + |w|_c = 0$ for q^0. When U is extended to $V = U \cup \{aa, ab\}$ by first adding aa and then ab, case (2) of Theorem 8 holds each time. $H_{q^0,V}$ has the basis $\{(1,0,0,0,0,0)^T, (0,1,0,-3,0,0)^T, (0,0,1,2,0,0)^T, (0,0,0,0,1,0)^T, (0,0,0,0,0,1)^T\}$. Extending V to $W = V \cup \{ac\}$, case (1) holds: $H_{q^0,W}$ has the basis $\{(1,0,0,0,0,0,0)^T, (0,1,0,-3,0,0,-3)^T, (0,0,1,2,0,0,0)^T, (0,0,0,0,1,0,-3)^T, (0,0,0,0,0,1,2)^T\}$, and we obtain a new invariant, $3|w|_a + 3|w|_{aa} - 2|w|_{ab} + |w|_{ac} = 0$.

We compute $H_{q,V}$ incrementally from $H_{q,U}$ as follows:

- for each basis vector ϕ, except for the initial unit vector $|\epsilon|_U$, we remember by which multiplication $F_a\psi$ it was obtained and how it was reduced; these steps are repeated for the new index v.
- we also remember which successors $F_a\psi$ are reduced to zero; when extending U by $v = u.a$, where $u \in U$, we check for all such ψ whether the reductions result in a nonzero vector, indicating that case (2) of Theorem 8 holds.

If case (2) holds for some location q, then the new basis vector η of H_q is invariant under all $F_{a,V}$, because, by choice, v cannot be a prefix of another sequence in V. Therefore, η is also contained in the subspace H_r for all locations r reachable from q. The check for case (2) therefore only needs to be performed in one location of each strongly connected component.

7 Conclusions and Future Work

We have introduced a new class of invariants, subsequence invariants, which are linear equalities over the occurrences of sequences of synchronization events. Subsequence invariants are a natural specification language for the description of the flow of synchronization events between different processes; basic equations over the number of occurrences of events as well their conditional and disjunctive combinations can easily be expressed.

The key advantage of subsequence invariants is that they can be computed individually for each process and compose by simple conjunction to invariants over the full system. The synthesis algorithms in this paper provide efficient means to obtain subsequence automatically from the process automata; they thus provide the foundation for a verification method that proves global system properties from locally obtained invariants.

A promising direction of future work is to extend the incremental invariant generation method from Section 6 into a refinement loop that automatically computes an appropriate set of subsequences. Also interesting is the idea of expanding the class of invariants by considering linear inequalities over the variables $|w|_U$. Such an approach could make use of established techniques for linear transition systems, combined with special properties of subsequence occurrences:

for example, Theorem 2 can be used to derive general, system-independent inequalities like $|w|_{aa} - |w|_a + |w|_\epsilon \geq 0$.

References

1. Bensalem, S., Lakhnech, Y.: Automatic generation of invariants. Formal Methods in System Design 15(1), 75–92 (1999)
2. Bjørner, N.S., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theoretical Comput. Sci. 173(1), 49–87 (1997)
3. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among the variables of a program. In: Proc. POPL, pp. 84–97 (January 1978)
5. Şerbănuţă, V.N., Şerbănuţă, T.F.: Injectivity of the Parikh matrix mappings revisited. Fundam. Inf. 73(1,2), 265–283 (2006)
6. Dräger, K., Finkbeiner, B.: Subsequence invariants. Technical Report 42, SFB/TR 14 AVACS (June 2008), http://www.avacs.org ISSN: 1860-9821
7. German, S.M., Wegbreit, B.: A Synthesizer of Inductive Assertions. IEEE transactions on Software Engineering 1(1), 68–75 (1975)
8. Sakarovitch, J., Simon, I.: Subwords. In: Lothaire, M. (ed.) Combinatorics on Words, pp. 105–144. Addison-Wesley, Reading (1983)
9. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. J. Mach. Learn. Res. 2, 419–444 (2002)
10. Mateescu, A., Salomaa, A., Yu, S.: Subword histories and Parikh matrices. J. Comput. Syst. Sci. 68(1), 1–21 (2004)
11. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
12. Parikh, R.J.: On context-free languages. J. ACM 13(4), 570–581 (1966)
13. Salomaa, A., Yu, S.: Subword conditions and subword histories. Inf. Comput. 204(12), 1741–1755 (2006)
14. Tiwari, A., Rueß, H., Saïdi, H., Shankar, N.: A technique for invariant generation. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 113–127. Springer, Heidelberg (2001)

Invariants for Parameterised Boolean Equation Systems*
(Extended Abstract)

Simona Orzan and Tim A.C. Willemse

Department of Mathematics and Computer Science,
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. The concept of invariance for Parameterised Boolean Equation Systems (PBESs) is studied in greater detail. We identify a weakness with the associated theory and fix this problem by proposing a stronger notion of invariance called global invariance. A precise correspondence is proven between the solution of a PBES and the solution of its invariant-strengthened version; this enables one to exploit global invariants when solving PBESs. Furthermore, we show that global invariants are robust w.r.t. all common PBES transformations and that the existing encodings of verification problems into PBESs preserve the invariants of the processes involved. These traits provide additional support for our notion of global invariants, and, moreover, provide an easy manner for transferring (e.g. automatically discovered) process invariants to PBESs. Several examples are provided that illustrate the advantages of using global invariants in various verification problems.

1 Introduction

Parameterised Boolean Equation Systems (PBESs), introduced in [10,9] as an extension of BESs [8] with data, and studied in detail in [7], provide a fundamental framework for studying and solving verification problems for complex reactive systems. Problems as diverse as model checking problems for symbolic transition systems [6] and real-time systems [16]; equivalence checking problems for a variety of process equivalences [2]; and static analysis of code [4] have been encoded in the PBES framework. The solution to the encoded problem can be found by solving the PBES. Several verification tools rely on PBESs or fragments thereof, e.g. the μCRL [6], mCRL2 [3] and the CADP [5] toolsets.

Solving a PBES is in general an undecidable problem, much like the problems that can be encoded in them. Nevertheless, there are pragmatic approaches to solving PBESs, such as *symbolic approximation* [7] and *instantiation* [3]. While these techniques have proved their merits in practice, the undecidability of solving PBESs in general implies that these techniques are not universally applicable.

* This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.602.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 187–202, 2008.

A concept that has turned out to be very powerful, especially when combined with symbolic approximation, is the notion of an *invariant* for PBESs. For instance, invariants have been used successfully in [2] when solving PBESs encoding the branching bisimulation problem for two systems: the invariants allowed the symbolic approximation process to terminate in a few steps, whereas there was no indication that it could have terminated without the invariant. As such, the notion of an invariant is a powerful tool which adds to the efficacy of techniques and tooling such as described in [6,7].

An invariant for a PBES, as defined in [7] (hereafter referred to as a *local invariant*), is a relation on data variables of a PBES that provides an over-approximation of the dependencies of the solution of a particular predicate variable X on its own domain.

We show that the theory of local invariants, as outlined in [7] does not allow for combining invariants with common solution-preserving PBES-manipulations; moreover, the theory cannot be extended to cope with such manipulations. We remedy this situation by introducing the concept of a *global invariant*, and show how this notion relates to local invariants. Moreover, we demonstrate that global invariants are preserved by the common PBES manipulation methods, viz. *unfolding*, *migration* and *substitution* [7]. An invariance theorem that allows one to calculate the solution for an equation system, using a global invariant to assist the calculation, is proved. As a side-result of the theory, we are able to provide a partial answer to a generalisation of an open problem coined in [7], which concerns the solution to a particular PBES pattern. Patterns are important as they allow for a simple *look-up* and *substitute* strategy to solving a PBES. Finally, we prove that traditional *process invariants* [1] are preserved under the PBES-encoding of the first-order modal μ-calculus model checking problem [6] and the PBES-encoding of various process equivalences [2].

Related Work. Invariants are indispensable in mature verification methodologies aiming at tackling complex cases, such as networks of parameterised systems [12,13], equivalence checks between reactive system [1] and for infinite data domains in general, such as hybrid systems [14]. Much research is aimed at stretching the limits of verification for specific classes of systems and properties. Techniques, such as invariants, that are developed for PBESs, on the other hand, are applicable to all problems that can be encoded in them.

Several works [12,14] focus on the automated discovery of invariants for specialised classes of specifications and properties. It is likely that these techniques can be adapted for specific PBESs. This is supported by our result that process invariants are preserved under the existing encodings of verification problems. An advantage of verification using PBESs is that predicates can be identified that are invariants for the PBES, but that fail to be invariants for the original process(es) involved. This is because the PBES-encoding incorporates more information from the input (see Section 5 for an example).

Structure. In Section 2, we introduce PBESs and some basic notation. We recall the definition of local invariants and introduce global invariants in Section 3,

and we show that the theory for local invariants has weaknesses, which are resolved by the theory for global invariants. The influence of solution-preserving manipulations for PBESs on global invariants is investigated in Section 4, and in Section 5, we investigate the relation between process invariants and global invariants. Two small examples illustrate several aspects of the developed theory. We present our conclusions in Section 7. All the proofs, more details and more examples can be found in the accompanying technical report [11].

2 Preliminaries

In this section, we give a brief overview of the concepts and notations that provide the basis to the theory in the remainder of this paper. We refer to [7] for a more detailed account.

Predicate formulae. Predicate formulae are part of the basic building blocks for PBESs; these are basically ordinary predicates extended with predicate variables.

Definition 1. *A* predicate formula *is a formula defined by the following grammar:*

$$\phi ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \forall d{:}D.\ \phi \mid \exists d{:}D.\ \phi \mid X(e)$$

where b is a data term of sort $\mathbb{B}$. Furthermore, X is a (sorted) predicate variable to which we associate a data variable d_X of sort D_X; e is a data term of the sort D_X. Data variables are taken from a set $\mathcal{D}$. The set of all predicate variables is referred to as $\mathcal{P}$.

The set of all predicate formulae is denoted Pred. Predicate formulae ϕ not containing predicate variables are referred to as *simple predicates*. The set of predicate variables that occur in a formula ϕ is denoted by $\mathsf{occ}(\phi)$. Note that negation does not occur in predicate formulae, except as an operator in data terms; $b \implies \phi$ is a shorthand for $\neg b \vee \phi$ for terms b of sort $\mathbb{B}$.

Note that we use predicate variables X to which we associate a single variable d_X of sort D_X instead of vectors $\boldsymbol{d_X}$ of sort $\boldsymbol{D_X}$. This does not incur a loss in generality; it is merely a matter of convenience.

Predicate formulae may contain both bound and unbound (free) data variables. We assume that the set of bound variables and the set of free variables in a predicate formula are disjoint. For a closed data term e, i.e. a data term not containing free data variables, we assume an interpretation function $[\![\_]\!]$ that maps the term e to the semantic data element $[\![e]\!]$ it represents. For open terms, we use a *data environment* ε that maps each variable from $\mathcal{D}$ to a data value of the intended sort. The interpretation of an open term e is denoted by $[\![e]\!]\varepsilon$ and is obtained in the standard way. We write $\varepsilon[e/d]$ to stand for the environment ε for all variables different from d, and $\varepsilon[e/d](d) = e$. A similar notation applies to predicate environments. For brevity, we do not explicitly distinguish between the abstract sorts of expressions and their semantic counterparts.

Definition 2. *Let θ be a* predicate environment *assigning a function $D_X \to \mathbb{B}$ to every predicate variable X, and let ε be a* data environment *assigning a value from domain D to every variable d of sort D. The interpretation $[\_]\theta\varepsilon$ of a predicate formula in the context of θ and ε is either true or false, as follows:*

$$
\begin{array}{ll}
[b]\,\theta\varepsilon & = [b]\,\varepsilon \\
[X(e)]\,\theta\varepsilon & = \textit{true iff } \theta(X)([e]\,\varepsilon) \\
[\forall d{:}D.\ \phi]\,\theta\varepsilon & = \textit{for all } v \in D,\ [\phi]\,\theta(\varepsilon[v/d]) \\
[\exists d{:}D.\ \phi]\,\theta\varepsilon & = \textit{for some } v \in D,\ [\phi]\,\theta(\varepsilon[v/d])
\end{array}
\qquad
\begin{array}{l}
[\phi_1 \wedge \phi_2]\,\theta\varepsilon = [\phi_1]\,\theta\varepsilon \textit{ and } [\phi_2]\,\theta\varepsilon \\
[\phi_1 \vee \phi_2]\,\theta\varepsilon = [\phi_1]\,\theta\varepsilon \textit{ or } [\phi_2]\,\theta\varepsilon
\end{array}
$$

Definition 3. *Let ϕ and ψ be predicate formulae. We write $\phi \to \psi$ iff for all predicate environments θ and all data environments ε,* $[\phi]\,\theta\varepsilon$ implies $[\psi]\,\theta\varepsilon$.

The symmetric closure of $\to$ induces a *logical equivalence* on Pred, denoted $\leftrightarrow$. Basic properties such as commutativity, idempotence and associativity of $\wedge$ and $\vee$ are immediately satisfied.

Predicate Variables and Substitution. A basic operation on predicate formulae is substitution of a predicate formula for a predicate variable. To this end, we introduce *predicate functions*: predicate formulae casted to functions. As a shorthand, we write $\phi_{\langle d_X \rangle}$ to indicate that ϕ is lifted to a function $(\lambda d_X{:}D_X.\ \phi)$, i.e. the variable d_X possibly occurring in ϕ acts as a placeholder for an expression of sort D_X. The semantics of a predicate function is defined in the context of a predicate environment θ and a data environment ε:

$$[\phi_{\langle d_X \rangle}]\,\theta\varepsilon \;=\; \lambda v{\in}D_X.\ [\phi]\,\theta\varepsilon[v/d_X]$$

The substitution of $\psi_{\langle d_X \rangle}$ for a predicate variable X in a predicate formula ϕ is defined by the following set of rules:

$$
\begin{array}{rcl}
b[\psi_{\langle d_X \rangle}/X] & = & b \\
Y(e)[\psi_{\langle d_X \rangle}/X] & = & \begin{cases} \psi[e/d_X] \text{ if } Y = X \\ Y(e) \quad \text{otherwise} \end{cases} \\
(\phi_1 \wedge \phi_2)[\psi_{\langle d_X \rangle}/X] & = & \phi_1[\psi_{\langle d_X \rangle}/X] \wedge \phi_2[\psi_{\langle d_X \rangle}/X] \\
(\phi_1 \vee \phi_2)[\psi_{\langle d_X \rangle}/X] & = & \phi_1[\psi_{\langle d_X \rangle}/X] \vee \phi_2[\psi_{\langle d_X \rangle}/X] \\
(\forall d{:}D.\ \phi)[\psi_{\langle d_X \rangle}/X] & = & \forall d{:}D.\ \phi[\psi_{\langle d_X \rangle}/X] \\
(\exists d{:}D.\ \phi)[\psi_{\langle d_X \rangle}/X] & = & \exists d{:}D.\ \phi[\psi_{\langle d_X \rangle}/X]
\end{array}
$$

Example 1. Consider the formulae $X(f(d)) \wedge Y(g(d))$ and $\psi := Y(h(d_Y))$. Then $(X(f(d)) \wedge Y(g(d)))[\psi_{\langle d_Y \rangle}/Y]$ yields: $X(f(d)) \wedge Y(h(g(d)))$. □

Property 1. Let ϕ, ψ be predicate formulae, and θ, ε environments. Then:

$$[\phi[\psi_{\langle d_X \rangle}/X]]\,\theta\varepsilon = [\phi]\,\theta[\,[\psi_{\langle d_X \rangle}]\,\theta\varepsilon\ /X]\varepsilon.$$ □

For convenience, we generalise single syntactic substitutions $\phi[\psi_{\langle d_X \rangle}/X]$ to finite sequences of substitutions using the following notation:

Definition 4. *Let $V = \langle X_1, \ldots, X_n \rangle$ be a vector of predicate variables and let ϕ_i be an arbitrary predicate formula. The* consecutive substitution $\phi \left[_{X_i \in V} \phi_{i\langle d_{X_i}\rangle}/X_i\right]$ *is defined as follows:*

$$\begin{cases} \phi \left[_{X_i \in \langle\rangle} \phi_{i\langle d_{X_i}\rangle}/X_i\right] & = \phi \\ \phi \left[_{X_i \in \langle X_1,\ldots,X_n\rangle} \phi_{i\langle d_{X_i}\rangle}/X_i\right] & = (\phi[\phi_{1\langle d_{X_1}\rangle}/X_1]) \left[_{X_i \in \langle X_2,\ldots,X_n\rangle} \phi_{i\langle d_{X_i}\rangle}/X_i\right] \end{cases}$$

When for all ϕ_i, at most variable X_i occurs in ϕ_i and all variables in $\langle X_1, \ldots, X_n\rangle$ are distinct, the consecutive substitution $\phi \left[_{X_i \in \langle X_1,\ldots,X_n\rangle} \phi_{i\langle d_{X_i}\rangle}/X_i\right]$ yields the same for all permutations of vector $\langle X_1, \ldots, X_n\rangle$, i.e. it behaves as a simultaneous substitution. In this case, we allow abuse of notation by writing $\phi \left[_{X_i \in \{X_1,\ldots,X_n\}} \phi_{i\langle d_{X_i}\rangle}/X_i\right]$.

Parameterised Boolean Equation Systems. A Parameterised Boolean Equation System (henceforth referred to as an *equation system*) is a finite sequence of equations of the form $(\sigma X(d_X{:}D_X) = \phi)$; ϕ is a predicate formula in which the variable d_X is considered bound. σ denotes either the least (μ) or the greatest (ν) fixed point. We denote the empty equation system by ϵ.

We say an equation system is *closed* whenever all predicate variables occurring at the right-hand side of an equation occur at the left-hand side of some equation. An equation system is *open* if it is not closed. For a given equation system $\mathcal{E}$, the set $\mathsf{bnd}(\mathcal{E})$ denotes the predicate variables occurring in the left-hand side of the equations of $\mathcal{E}$, and the set $\mathsf{occ}(\mathcal{E})$ denotes the set of predicate variables occurring in the predicate formulae of the equations of $\mathcal{E}$. The *solution* to an equation system is a predicate environment, defined as follows:

Definition 5. *Given a predicate environment θ and an equation system $\mathcal{E}$. The* solution $[\![\mathcal{E}]\!]\theta\varepsilon$ *is an environment that is defined as follows:*

$$\begin{aligned} [\![\epsilon]\!]\theta\varepsilon &= \theta \\ [\![(\sigma X(d_X{:}D_X) = \phi)\mathcal{E}]\!]\theta\varepsilon &= [\![\mathcal{E}]\!](\theta\Big[\sigma\mathcal{X}{\in}[D_X \to \mathbb{B}].\ [\![\phi_{\langle d_X\rangle}]\!]([\![\mathcal{E}]\!]\theta[\mathcal{X}/X])\varepsilon/X\Big])\varepsilon \end{aligned}$$

Note that the fixed points are taken over the complete lattice of functions $([D_X \to \mathbb{B}], \sqsubseteq)$ for (possibly infinite) data sets D_X, where $f \sqsubseteq g$ is defined as the point-wise ordering: $f \sqsubseteq g$ iff for all $v \in D_X$: $f(v)$ implies $g(v)$. The predicate transformer associated to a predicate function $[\![\phi_{\langle d_X\rangle}]\!]\theta\varepsilon$, denoted

$$\lambda\mathcal{X}{\in}[D_X \to \mathbb{B}].\ [\![\phi_{\langle d_X\rangle}]\!]\theta[\mathcal{X}/X]\varepsilon$$

is a monotone operator [6,7]. The existence of fixed points of this operator in the lattice $([D_X \to \mathbb{B}], \sqsubseteq)$ follows immediately from Tarski's Theorem [15].

Note 1. The solution to an equation system is sensitive to the ordering of the equations: while $(\mu X = Y)(\nu Y = X)$ has $\bot$ as solution for X and Y, the equation system $(\nu Y = X)(\mu X = Y)$ has $\top$ as solution for X and Y. Manipulations such as unfolding, migration and substitution, however, do not affect the solution to

an equation system [7]. Using the latter two, all equation systems can be solved (using a strategy called *Gauß Elimination*), *provided* that one has the techniques and tools to eliminate a predicate variable from its defining equation. One such methods is e.g. *symbolic approximation*, see [7].

3 Invariants for Equation Systems

Invariants for equation systems first appeared in [7]. We first repeat its definition:

Definition 6. *Let* $(\sigma X(d_X{:}D_X) = \phi)$ *be an equation and let* I *be a simple predicate formula, i.e. a formula without predicate variable occurrences. Then* I *is an invariant of* X *iff*

$$I \wedge \phi \;\leftrightarrow\; (I \wedge \phi)[(I \wedge X(d_X))_{\langle d_X \rangle}/X]$$

The above definition may appear awkward to those familiar only with invariants for transition systems. It does, however, express what is normally understood as the invariance property; the unusual appearance is a consequence of the possibility of having multiple occurrences of X in subformulae of ϕ. The invariance criterion only concerns a transfer property on equation systems: an initialisation criterion is not applicable at this level. The analogue to the initialisation criterion is, however, part of Theorem 2 (see page 194), and Theorems 40 and 42 of [7]. For completeness' sake, we recall the latter and expose its weakness by an example:

Theorem 1 (See [7]). *Let* $(\sigma X(d_X{:}D_X) = \phi)$ *be an equation and let* I *be an invariant of* X*. Assume that:*

1. *for some* χ *with* $X \notin \mathsf{occ}(\chi)$*, we have for all equation systems* $\mathcal{E}$ *and all* η, ε*:* $[\![(\sigma X(d_X{:}D_X) = I \wedge \phi)\ \mathcal{E}]\!]\eta\varepsilon = [\![(\sigma X(d_X{:}D_X) = \chi)\ \mathcal{E}]\!]\eta\varepsilon$.
2. *for the predicate formula* ψ *we have* $\psi \leftrightarrow \psi[(I \wedge X(d_X))_{\langle d_X \rangle}/X]$.

Then for all equation systems $\mathcal{E}_0, \mathcal{E}_1$ *and all environments* η, ε*:*

$$\begin{aligned} &[\![(\sigma' Y(d_Y{:}D_Y) = \psi)\ \mathcal{E}_0(\sigma X(d_X{:}D_X) = \phi)\ \mathcal{E}_1]\!]\eta\varepsilon \\ = \;&[\![(\sigma' Y(d_Y{:}D_Y) = \psi[\chi_{\langle d_X \rangle}/X])\ \mathcal{E}_0(\sigma X(d_X{:}D_X) = \phi)\ \mathcal{E}_1]\!]\eta\varepsilon. \end{aligned}$$ □

Theorem 1 states that if one can show that $\psi \leftrightarrow \psi[(I \wedge X(d_X))_{\langle d_X \rangle}/X]$ (the analogue to the initialisation criterion for an invariant), and χ is the solution of X's equation strengthened with I, then it suffices to solve Y using χ for X rather than X's original solution. However, a computation of χ cannot take advantage of PBES manipulations when X's equation is *open*. Such equations arise when encoding process equivalences [2] and model checking problems [9,6]. A second issue is that invariants may "break" as a result of a substitution:

Example 2. Consider the following (constructed) closed equation system:

$$\begin{aligned} &(\mu X(n{:}\mathbb{N}) = n \geq 2 \wedge Y(n)) \\ &(\mu Y(n{:}\mathbb{N}) = Z(n) \vee Y(n+1)) \\ &(\mu Z(n{:}\mathbb{N}) = n < 2 \vee Y(n-1)) \end{aligned} \tag{1}$$

The simple predicate formula $n \geq 2$ is an invariant for equation Y in equation system (1): $n \geq 2 \wedge (Z(n) \vee Y(n+1)) \;\leftrightarrow\; n \geq 2 \wedge (Z(n) \vee (n+1 \geq 2 \wedge Y(n+1)))$. However, substituting $n < 2 \vee Y(n-1)$ for Z in the equation of Y in system (1) yields the equation system of (2):

$$\begin{array}{l}(\mu X(n{:}\mathbb{N}) = n \geq 2 \wedge Y(n)) \\ (\mu Y(n{:}\mathbb{N}) = n < 2 \vee Y(n-1) \vee Y(n+1)) \\ (\mu Z(n{:}\mathbb{N}) = n < 2 \vee Y(n-1))\end{array} \tag{2}$$

The invariant $n \geq 2$ of Y in (1) fails to be an invariant for Y in (2). Worse still, computing the solution to Y without relying on the equation for Z leads to an awkward approximation process that does not terminate; one has to resort to using a pattern to obtain the solution to equation Y of (1):

$$(\mu Y(n{:}\mathbb{N}) = n \geq 2 \wedge \exists i{:}\mathbb{N}.\; Z(n+i))$$

Using this solution for Y in the equation for X in (1), and solving the resulting equation system leads to the solution $\lambda v{\in}\mathbb{N}.\; v \geq 2$ for X and $\lambda v{\in}\mathbb{N}.\; \top$ for Y and Z. A weakness of Theorem 1 is that in solving the invariant-strengthened equation for Y, one cannot employ knowledge about the equation system at hand as this is prevented by the strict conditions of Theorem 1. Weakening these conditions to incorporate information about the actual equation system is impossible without affecting correctness: solving, e.g., the invariant-strengthened version for Y of (2) leads to the solution $\lambda v{\in}\mathbb{N}.\; \bot$ for X. Theorem 40 of [7] is ungainly as it even introduces extra equations. □

Example 2 shows that identified invariants (cf. [7]) fail to remain invariants when substitution is exercised on the equation system, and, more importantly, that Theorem 1 cannot employ PBES manipulations for simplifying the invariant-strengthened equation.

As we demonstrate in this paper, both issues can be remedied by using a slightly stronger invariance criterion, taking all predicate variables of an equation system into account. This naturally leads to a notion of *global invariance*; in contrast, we refer to the type of invariance defined in Def. 6 as *local invariance*.

Let $f{:}V \to \mathsf{Pred}$, $V \subseteq \mathcal{P}$, be a function that maps a predicate variable to a predicate formula. We say f is *simple* iff $f(X)$ is simple, that is not containing predicate variables, for all $X{\in}V$. Note that the notation $f(X)$ is purely *meta-notation*; e.g. it is not affected by syntactic substitutions: $f(X)[\psi_{\langle d_X \rangle}/X]$ remains $f(X)$, since $f(X)$ is simple.

Definition 7. *The simple function $f{:}V \to \mathsf{Pred}$ is said to be a* global invariant *for an equation system $\mathcal{E}$ iff $V \supseteq \mathsf{bnd}(\mathcal{E})$ and for each $(\sigma X(d_X{:}D_X) = \phi)$ occurring in $\mathcal{E}$, we have:*

$$f(X) \wedge \phi \;\leftrightarrow\; (f(X) \wedge \phi)\left[_{X_i \in V}(f(X_i) \wedge X_i(d_{X_i}))_{\langle d_{X_i} \rangle}/X_i\right].$$

Proposition 1. *Let $f{:}V \to \mathsf{Pred}$ be a global invariant for an equation system $\mathcal{E}$. Let $W \subseteq V$. Then for all equations $(\sigma X(d_X{:}D_X) = \phi)$ in $\mathcal{E}$, we have:*

$$f(X) \wedge \phi \;\leftrightarrow\; (f(X) \wedge \phi)\left[_{X_i \in W}(f(X_i) \wedge X_i(d_{X_i}))_{\langle d_{X_i} \rangle}/X_i\right]. \qquad \square$$

Corollary 1. *Let f be a global invariant for an equation system $\mathcal{E}$. Then $f(X)$, for any $X \in \mathsf{bnd}(\mathcal{E})$ is a local invariant.* □

Note 2. The reverse of the above corollary does not hold: if for all $X \in \mathsf{bnd}(\mathcal{E})$, we have a predicate formula $f(X)$ that is a local invariant for X in $\mathcal{E}$, then f is *not* necessarily a global invariant. Consider the following equation system:

$$(\nu X(n{:}\mathbb{N}) = Y(n-1))\ (\mu Y(n{:}\mathbb{N}) = X(n+1))$$

X and Y both have $n \geq 5$ as local invariants (in fact, any simple predicate is a local invariant), but $(\lambda Z \in \{X, Y\}.\ n \geq 5)$ fails to be a global invariant.

Let $\mathsf{pvi}(\phi)$ yield the set of *predicate variable instantiations* in ϕ:

$$\begin{array}{ll} \mathsf{pvi}(b) = \emptyset & \mathsf{pvi}(X(e)) = \{X(e)\} \\ \mathsf{pvi}(\forall d{:}D.\ \phi) = \mathsf{pvi}(\phi) & \mathsf{pvi}(\phi_1 \wedge \phi_2) = \mathsf{pvi}(\phi_1) \cup \mathsf{pvi}(\phi_2) \\ \mathsf{pvi}(\exists d{:}D.\ \phi) = \mathsf{pvi}(\phi) & \mathsf{pvi}(\phi_1 \vee \phi_2) = \mathsf{pvi}(\phi_1) \cup \mathsf{pvi}(\phi_2) \end{array}$$

A sufficient condition for a function f to be a global invariant is given below:

Property 2. Let $\mathcal{E}$ be an equation system and $f{:}\mathsf{bnd}(\mathcal{E}) \to \mathsf{Pred}$ a simple function; then f is a global invariant for $\mathcal{E}$ if for every equation $(\sigma X(d_X{:}D_X) = \phi)$ in $\mathcal{E}$ we have $f(X) \to \bigwedge_{Y(e) \in \mathsf{pvi}(\phi)} (f(Y))[e/d_Y]$. □

We next establish an exact correspondence between the solution of an equation system $\mathcal{E}$ and the equation system $\mathcal{E}'$ which is derived from $\mathcal{E}$ by strengthening it with the global invariant. Strengthening is achieved by the operation Apply:

Definition 8. *Let $f{:}V \to \mathsf{Pred}$ be a simple function. Let $\mathcal{E}$ be an equation system satisfying $\mathsf{bnd}(\mathcal{E}) \subseteq V$. The equation system $\mathit{Apply}\ (f, \mathcal{E})$ is defined as follows:*

$$\begin{array}{ll} \mathit{Apply}\ (f, \epsilon) & = \epsilon \\ \mathit{Apply}\ (f, (\sigma X(d_X{:}D_X) = \phi)\ \mathcal{E}_0) & = (\sigma X(d_X{:}D_X) = f(X) \wedge \phi)\ \mathit{Apply}\ (f, \mathcal{E}_0) \end{array}$$

The correctness of the above-mentioned correspondence relies, among others, on the following lemma. The main result of this section is Theorem 2, which improves upon Theorem 1; it immediately follows the below lemma.

Lemma 1. *Let $(\sigma X(d_X{:}D_X) = \phi)$ be a possibly open equation. Let $f{:}V \to \mathsf{Pred}$ be a simple function such that*

1. $\mathsf{occ}(\phi) \subseteq V$.
2. $f(X) \wedge \phi \leftrightarrow (f(X) \wedge \phi)[(f(X) \wedge X(d_X))_{\langle d_X \rangle}/X]$.

Then for all environments η, ε:

$$\begin{array}{l} \lambda v{\in}D_X.\ [\![f(X)]\!]\varepsilon[v/d_X] \wedge (\sigma \mathcal{X} \in [D_X \to \mathbb{B}].\ [\![\phi_{\langle d_X \rangle}]\!]\eta[\mathcal{X}/X]\varepsilon)(v) \\ = \lambda v{\in}D_X.\ [\![f(X)]\!]\varepsilon[v/d_X] \wedge (\sigma \mathcal{X} \in [D_X \to \mathbb{B}].\ [\![(f(X) \wedge \phi)_{\langle d_X \rangle}]\!]\eta[\mathcal{X}/X]\varepsilon)(v). \end{array}$$ □

Theorem 2. *Let $f{:}V \to \mathsf{Pred}$ be a simple function. Let $\mathcal{E}$ be an equation system and let η_1, η_2 be arbitrary predicate environments. If the following holds:*

1. $\mathsf{bnd}(\mathcal{E}) \cup \mathsf{occ}(\mathcal{E}) \subseteq V$ *and*
2. *for all* $X \in V$*:*
   (a) $[\![f(X) \wedge X(d_X)]\!]\eta_1\varepsilon = [\![f(X) \wedge X(d_X)]\!]\eta_2\varepsilon$.
   (b) $f(X) \wedge \phi \leftrightarrow (f(X) \wedge \phi)\left[_{X_i \in V}(f(X_i) \wedge X_i(d_{X_i}))_{\langle d_{X_i}\rangle}/X_i\right]$.

then we have for all $X \in V$*:*

$$[\![f(X) \wedge X(d_X))]\!]\,([\![\mathcal{E}]\!]\,\eta_1\varepsilon)\varepsilon = [\![f(X) \wedge X(d_X))]\!]\,([\![\mathit{Apply}\ (f, \mathcal{E})]\!]\,\eta_2\varepsilon)\varepsilon. \qquad \square$$

Corollary 2. *Let* $f{:}V \to \mathsf{Pred}$ *be a global invariant for an equation system* $\mathcal{E}$*. Then for all predicate formulae* ϕ *with* $\mathsf{occ}(\phi) \subseteq V$ *and all environments* η, ε*:*

$$\begin{aligned} &\phi \leftrightarrow \phi\left[_{X_i \in V}(f(X_i) \wedge X_i(d_{X_i}))_{\langle d_{X_i}\rangle}/X_i\right] \\ &\textit{implies}\ [\![\phi]\!]\,([\![\mathcal{E}]\!]\,\eta\varepsilon)\varepsilon = [\![\phi]\!]\,([\![\mathit{Apply}\ (f, \mathcal{E})]\!]\,\eta\varepsilon)\varepsilon \end{aligned} \qquad \square$$

This means that for an equation system $\mathcal{E}$ and a global invariant f of $\mathcal{E}$, it does not matter whether we use $\mathcal{E}$ or its invariant-strengthened version $\mathsf{Apply}\ (f, \mathcal{E})$ to evaluate a predicate formula ϕ for which the initialisation criterion for invariant f holds. As another consequence of Theorem 2, we have the proposition below:

Proposition 2. *Let* $\mathcal{E}$ *be an equation system. Let* f *be a global invariant for* $\mathcal{E}$ *and assume* $\mathcal{E}$ *contains an equation for* X *of the form:*

$$(\nu X(d{:}D) = f(X) \wedge \bigwedge_{i \in I} \mathsf{Q}_1\, e_i^1{:}E_i^1 \ldots \mathsf{Q}_{m_i}\, e_i^{m_i}{:}E_i^{m_i}.\ \psi_i \implies X(g_i(d, e_i^1, \ldots, e_i^{m_i})))$$

where $\mathsf{Q}_j \in \{\forall, \exists\}$ *for any* j*, and for all* i*,* ψ_i *are simple predicate formulae and* g_i *is a data term that depends only on the values of* d *and* $e_i^1, \ldots, e_i^{m_i}$*. Then* X *has the solution* $f(X)$*.* $\square$

In the terminology of [7], the equation above is a pattern which has solution $f(X)$. This pattern is an instance of a generalisation of the unsolved pattern of [7]. This pattern turns out to be extremely useful in the examples of Section 6.

4 Preservation of Global Invariants under Solution-Preserving Manipulations

A serious defect of local invariants is that this notion is not robust with respect to *substitution*. In this section, we study the robustness of (global) invariants with respect to most common solution-preserving manipulations, viz. *migration*, *unfolding* and *substitution* [7].

Theorem 3. *Let* $\mathcal{E} :\equiv \mathcal{E}_0\ (\sigma X(d_X{:}D_X) = \phi)\ \mathcal{E}_1\ \mathcal{E}_2$ *be an equation system with* $\mathsf{occ}(\phi) = \emptyset$*. Let* $\mathcal{F} :\equiv \mathcal{E}_0\ \mathcal{E}_1\ (\sigma X(d_X{:}D_X) = \phi)\ \mathcal{E}_2$ *be the result of a* migration*. If* $f{:}V \to \mathsf{Pred}$ *is a global invariant for* $\mathcal{E}$ *then* f *is a global invariant for* $\mathcal{F}$*.* $\square$

Unfolding and substitution [7] involve replacing predicate variables with the right-hand side expressions of their corresponding equation. The difference is that unfolding operates locally and substitution is a global operation. The following lemma proves robustness of invariants under replacing variables with their corresponding right-hand side expressions.

Lemma 2. *Let $\mathcal{E}$ be an equation system and let $f{:}V \to \mathsf{Pred}$ be a global invariant for $\mathcal{E}$. For any predicate variable $X \in \mathsf{bnd}(\mathcal{E})$, we denote the right-hand side of X's defining equation in $\mathcal{E}$ by ϕ_X. Then for all $X, Y \in \mathsf{bnd}(\mathcal{E})$:*

$$\begin{array}{l} f(X) \wedge \phi_X[\phi_{Y\,\langle d_Y\rangle}/Y] \\ \leftrightarrow (f(X) \wedge \phi_X[\phi_{Y\,\langle d_Y\rangle}/Y])\ \left[_{Z\in V}(f(Z) \wedge Z(d_Z))_{\langle d_Z\rangle}/Z\right] \end{array} \qquad \square$$

The above lemma immediately leads to the robustness of invariants under substitution and unfolding. This is expressed by the following theorems:

Theorem 4. *Let $\mathcal{E} :\equiv \mathcal{E}_0\ (\sigma X(d_X{:}D_X) = \phi)\ \mathcal{E}_1$ be an equation system and let $f{:}V \to \mathsf{Pred}$ a global invariant for $\mathcal{E}$. Then f is also a global invariant for the equation system $\mathcal{F} :\equiv \mathcal{E}_0\ (\sigma X(d_X{:}D_X) = \phi[\phi_{\langle d_X\rangle}/X])\ \mathcal{E}_1$.* $\square$

Theorem 5. *Let $\mathcal{E} :\equiv \mathcal{E}_0\ (\sigma X(d_X{:}D_X) = \phi)\ \mathcal{E}_1\ (\sigma' Y(d_Y{:}D_Y) = \psi)\ \mathcal{E}_2$ and $\mathcal{F} :\equiv \mathcal{E}_0\ (\sigma X(d_X{:}D_X) = \phi[\psi_{\langle d_Y\rangle}/Y])\ \mathcal{E}_1\ (\sigma' Y(d_Y{:}D_Y) = \psi)\ \mathcal{E}_2$ be PBESs. If $f{:}V \to \mathsf{Pred}$ is a global invariant for $\mathcal{E}$ then f is also a global invariant for $\mathcal{F}$.* $\square$

An interesting observation is that both substitution and unfolding not only preserve existing global invariants, but also may lead to *new* global invariants. We illustrate this phenomenon with an example for unfolding.

Example 3. Let $\nu X(n{:}\mathbb{N}) = X(n+1)$ be an equation system. Using unfolding, we obtain the equivalent equation system $\nu X(n{:}\mathbb{N}) = X(n+2)$. Clearly, the function f that assigns to X the predicate formula $\mathsf{even}(n)$ is a global invariant for the latter equation. However, f is not a global invariant for the original equation. Thus, by unfolding, the set of invariants for an equation system may increase. $\square$

5 Process Invariants and Equation Invariants

Linear process equations (LPEs) have been proposed as *symbolic* representations of general (infinite) labelled transition systems, a semantic framework for specifying and analysing complex, reactive systems. In an LPE, the state of a process is modelled by a finite vector of (possibly infinite) sorted variables, and the behaviour is described by a finite set of condition-action-effect rules. The apparent restrictiveness of this format is misleading: parallelism, (infinite) non-determinism and other operators can often be mapped losslessly onto LPEs.

Definition 9. *A* linear process equation *is an equation taking the form*

$$P(d{:}D) = \sum\{\sum_{e_a:E_a} c_a(d,e_a) \Longrightarrow a(f_a(d,e_a)) \cdot P(g_a(d,e_a)) \mid a \in \mathcal{A}ct\}$$

where $f_a{:}D \times E_a \to D_a$, $g_a{:}D \times E_a \to D$ and $c_a{:}D \times E_a \to \mathbb{B}$ for each action label $a \in \mathcal{A}ct$. D, D_a and E_a are general data sorts. The restrictions to single sorts D and E_a are again only for brevity and do not incur a loss of generality.

In the above definition, the LPE P specifies that if in the current state d the condition $c_a(d, e_a)$ holds, for an arbitrary e_a of sort E_a, then an action a carrying data parameter $f_a(d, e_a)$ is possible and the effect of executing this action is that the state is changed to $g_a(d, e_a)$. Thus, the values of the condition, action parameter and new state may depend on the current state and the non-deterministic chosen value for variable e_a.

Definition 10. *Let P be the LPE of Def. 9. A simple predicate ι is an* invariant *for P iff for all actions $a \in \mathcal{A}ct$: $\iota \wedge c_a(d, e_a) \rightarrow (\iota[g_a(d, e_a)/d])$ holds.*

Model Checking. In [9,6], the *first-order modal μ-calculus* (μ-calculus for short) is defined, a modal language for verification of data-dependent process specifications. The language is a first-order extension of the standard modal μ-calculus due to Kozen. It permits the use of data variables and parameters to capture the essential data-dependencies in the process behaviour. The grammar of the calculus is given by the following rules:

$$\phi ::= b \mid X(e) \mid \neg\phi \mid \phi \wedge \phi \mid \forall d{:}D.\ \phi \mid [\alpha]\phi \mid (\nu X(d_f{:}D_f := e).\ \phi)$$
$$\alpha ::= b \mid a(e) \mid \neg\alpha \mid \alpha \wedge \alpha \mid \forall d{:}D.\alpha$$

where ν is the greatest fixed point sign (note that $\mu X(d_X{:}D_X := e).\phi$ is a shorthand for $\neg\nu X(d_X{:}D_X := e).\neg\phi[\neg X(d_X)_{\langle d_X \rangle}/X]$). The meaningful formulae are those formulae for which every occurrence of a variable X is under an even number of negations. The semantics of μ-calculus formulae is defined over an LTS, induced by an LPE P, see [6,9] for details. The global model checking problem $P \models \Phi$ and the local model checking problem $P(e) \models \Phi$, where e is an initial value for P and Φ is a μ-calculus formula, can be translated to the problem of solving the equation system $\mathbf{E}(\Phi)$ [9,6].

Theorem 6. *Let Φ be a μ-calculus formula. Let ι be an invariant for the LPE P of Def. 9. Then $(\lambda X{\in}\mathsf{bnd}(\mathbf{E}(\Phi)).\ \iota)$ is a global invariant of $\mathbf{E}(\Phi)$.* □

The reverse of the above theorem does not hold: if f is a global invariant for an equation system $\mathbf{E}(\Phi)$ for some formula Φ and LPE P, then f does not necessarily lead to an invariant for the process P (see the below example).

Example 4. Consider the following process, that models the rise and fall of a stock value of a company and may report its current value if asked.

$$\begin{aligned} M(v{:}\mathbb{N}) = &\textstyle\sum_{m:\mathbb{N}} \mathtt{up} \cdot M(v+m) \\ &+ \mathtt{current}(v) \cdot M(v) \\ &+ \textstyle\sum_{m:\mathbb{N}} m \leq v \implies \mathtt{down} \cdot M(v-m) \end{aligned}$$

Verifying that without decreases, the stock value is always above threshold T (provided it is so initially), i.e. $\nu X.[\neg\mathtt{down}]X \wedge \forall n{:}\mathbb{N}.[\mathtt{current}(n)](n > T)$, using an equation system boils down to solving the below equation system:

$$\nu X(v{:}\mathbb{N}) = (\forall m{:}\mathbb{N}.\ X(v+m)) \wedge \forall n{:}\mathbb{N}.\ v = n \implies n > T$$

Clearly, X has $v{>}T$ as an invariant whereas this is not an invariant for M. □

Process Equivalences. In [2] various equivalences, such as strong and branching bisimulation, between two LPEs M and S have been encoded as solution problems of equation systems. Branching bisimulation is the most complex of the process equivalences tackled in [2], yielding the equation system $\nu E_2\ \mu E_1$, which is of alternation depth 2. Here, E_1 and E_2 are sets of equations obtained from a syntactic manipulation of the input LPEs M and S, where $\mathsf{bnd}(E_2) = \{X^{M,S}, X^{S,M}\}$ and $\mathsf{bnd}(E_1) = \{Y_{\mathtt{a}}^{M,S}, Y_{\mathtt{a}}^{S,M} \mid \mathtt{a} \in \mathcal{A}ct\}$.

Theorem 7. *Let M be an LPE. Assume ι is an invariant for LPE M. We define function f as follows:*

$$f(Z) = \begin{cases} \iota & \text{if } Z \in \{X^{M,S}, X^{S,M}, Y_{\mathtt{a}}^{S,M} \mid \mathtt{a} \in \mathcal{A}ct\} \\ \iota \wedge c_{\mathtt{a}}^{M}(d,e) & \text{if } Z \in \{Y_{\mathtt{a}}^{M,S} \mid \mathtt{a} \in \mathcal{A}ct\} \end{cases}$$

Then f is a global invariant for the equation system $\nu E_2 \mu E_1$, resulting from the encoding of branching bisimulation between M and a second LPE S. □

The remaining encodings in [2] yield similar global invariants, see [11]. The significance of the preservation of process invariants under the PBES-encoding of an equivalence lies in the fact that this helps ensuring that the solution of the equation system does not relate all unreachable states of the input processes. Relating unreachable parts of processes is often neither meaningful nor computationally tractable (in particular for infinite state systems).

6 Examples

To illustrate how invariants typically assist in solving equation systems, we provide two easily understood examples of verifications using equation systems. The first example treats the privacy problem of a rudimentary voting protocol; the second is a mutual exclusion problem for readers and writers.

6.1 Voting Protocol

The voting protocol is given by the LPE E below. The intended votes of participants are modelled by variable V, a bitlist; we write $V.i$ to indicate the vote of voter i. A high bit represents a *yes* and a low bit represents a *no* vote. Registered voters are maintained in set R and parameters y, n record the number of casted yes/no votes so far. Voting of a person is modelled by action `vote`, and it follows no particular order. The outcome of the vote is published by action `outcome`.

$$\begin{aligned} E(V{:}\mathcal{L}(\{0,1\}), R{:}2^{\mathbb{N}}, y, n{:}\mathbb{N}) = \\ \quad R = \emptyset \Longrightarrow \mathtt{outcome}(y,n) \cdot \delta \\ + \textstyle\sum_{i:\mathbb{N}}\ i \in R \Longrightarrow \mathtt{vote}(i) \cdot E(V, R \setminus \{i\}, y + V.i, n + (1 - V.i)) \end{aligned}$$

One way to warrant privacy of the voting process is to ensure that an external observer cannot tell whether $V.i = 0$ or $V.i = 1$ for any voter i. Formally, privacy is then guaranteed if process $E(l, r, 0, 0)$ is strongly bisimilar to $E(\pi(l), r, 0, 0)$,

where list $\pi(l)$ is an arbitrary permutation of list l. Strong bisimilarity is encoded by the below equation system (see [2] for the general translation scheme).

$$
\begin{aligned}
&(\nu X(V{:}\mathcal{L}(\{0,1\}), R{:}2^{\mathbb{N}}, y, n{:}\mathbb{N}, V'{:}\mathcal{L}(\{0,1\}), R'{:}2^{\mathbb{N}}, y', n'{:}\mathbb{N}) = \\
&\quad (\forall i{:}\mathbb{N}.\ i \in R \implies (i \in R' \\
&\qquad \wedge X(V, R \setminus \{i\}, y + V.i, n + (1 - V.i), V', R' \setminus \{i\}, y' + V'.i, n' + (1 - V'.i)))) \\
&\wedge (\forall i{:}\mathbb{N}.\ i \in R' \Rightarrow (i \in R \\
&\qquad \wedge X'(V, R \setminus \{i\}, y + V.i, n + (1 - V.i), V', R' \setminus \{i\}, y' + V'.i, n' + (1 - V'.i)))) \\
&\wedge (R = \emptyset \iff R' = \emptyset) \wedge (R = \emptyset \implies (y = y' \wedge n = n'))) \\
&(\nu X'(V'{:}\mathcal{L}(\{0,1\}), R'{:}2^{\mathbb{N}}, y', n'{:}\mathbb{N}, V{:}\mathcal{L}(\{0,1\}), R{:}2^{\mathbb{N}}, y, n{:}\mathbb{N}) = \\
&\quad X(V, R, y, n, V', R', y', n'))
\end{aligned}
$$

$E(l, r, 0, 0)$ and $E(\pi(l), r, 0, 0)$ are bisimilar iff $X(l, r, 0, 0, \pi(l), r, 0, 0)$ is true. A symbolic approximation of variable X generates a non-converging series of increasingly complex equations expressing constraints on subsets of R, meaning that we cannot compute the general solution to X.

The equation system encodes the strong bisimulation relation between two processes E, i.e. both reachable and unreachable states of the two processes E will be related in the solution to X. However, we are interested only in the answer to $X(l, r, 0, 0, \pi(l), r, 0, 0)$. We state the following three predicate formulae:

- $\iota_1 := R = R'$ formalises that we are not interested in relating information for different sets of voters,
- $\iota_2 := y + n = y' + n'$ formalises that the total number of expressed votes should be the same in both protocols,
- $\iota_3 := y + \sum_{i \in R} V.i = y' + \sum_{i \in R'} V'.i$ formalises, among others, that we are dealing with permutations.

Let $\iota := \iota_1 \wedge \iota_2 \wedge \iota_3$; from Property 2, we immediately conclude that ι is an invariant for X and X'. Note that ι is a tautology when instantiated with the initial values due to the verification problem $E(l, r, 0, 0) = E(\pi(l), r, 0, 0)$. So, without affecting the answer to our verification problem, we can strengthen the equations for X and X' with ι. The variable X', appearing in the equation for X can be removed by a substitution. We observe that for equation X:

$$(\iota \wedge (R = \emptyset \iff R' = \emptyset) \wedge (R = \emptyset \implies (y = y' \wedge n = n'))) \iff \iota$$

We find that the equation for X is of the form of Proposition 2; it therefore has solution ι. Since $X(l, r, 0, 0, \pi(l), r, 0, 0)$ holds, privacy is indeed guaranteed.

6.2 Readers-Writers Mutual Exclusion

We consider a standard mutual exclusion problem between distributed *readers* and *writers*. A total of $N > 0$ (N is some arbitrary value) readers and writers are assumed.

$$
\begin{aligned}
P(n_r, n_w, t{:}\mathbb{N}) = {} & t \geq 1 \implies \mathtt{r}_s \cdot P(n_r + 1, n_w, t - 1) \\
& + n_r > 0 \implies \mathtt{r}_e \cdot P(n_r - 1, n_w, t + 1) \\
& + t \geq N \implies \mathtt{w}_s \cdot P(n_r, n_w + 1, t - N) \\
& + n_w > 0 \implies \mathtt{w}_e \cdot P(n_r, n_w - 1, t + N)
\end{aligned}
$$

Here the actions $\mathtt{r}_s$ and $\mathtt{w}_s$ express the starting of reading and writing of a process. Likewise, the actions $\mathtt{r}_e$ and $\mathtt{w}_e$ model the ending of reading and writing. Mutual exclusion between readers and writers holds when:

1. No writer can start if readers are reading: $\nu X.[\top]X \wedge [\mathtt{r}_s]\nu Y.([\neg \mathtt{r}_e]Y \wedge [\mathtt{w}_s]\bot)$.
2. No reader can start if writers are busy: $\nu X.[\top]X \wedge [\mathtt{w}_s]\nu Y.([\neg \mathtt{w}_e]Y \wedge [\mathtt{r}_s]\bot)$.

We only treat the first property; proving the second property follows the same reasoning. The equation system that encodes the first property is given below:

$$\begin{array}{l}(\nu X(n_r, n_w, t{:}\mathbb{N}) = ((t \geq 1 \implies (X(n_r + 1, n_w, t-1) \wedge Y(n_r + 1, n_w, t-1))) \\ \quad \wedge (n_r > 0 \implies X(n_r - 1, n_w, t+1)) \wedge (t \geq N \implies X(n_r, n_w + 1, t - N)) \\ \quad\quad \wedge (n_w > 0 \implies X(n_r, n_w - 1, t + N)))) \\ (\nu Y(n_r, n_w, t{:}\mathbb{N}) = t < N \wedge (t \geq 1 \implies Y(n_r + 1, n_w, t-1)) \\ \quad \wedge (n_w > 0 \implies Y(n_r, n_w - 1, t + N)))\end{array}$$

With standard techniques, Y can only be solved using an unwieldy pattern [7], which introduces multiple quantifications and additional selector functions; symbolic approximation does not converge in a finite number of steps. The use of invariants is the most appropriate strategy here. An invariant of process P is $t = N - (n_r + n_w \cdot N)$, which, by Theorem 6 is also a global invariant for the equations X and Y. Furthermore, $n_r \geq 1$ for Y and $\top$ for X is a global invariant. Both X and Y can be strengthened with the above invariants. The simple predicate formula $t < N$ follows from $t = N - (n_r + n_w \cdot N) \wedge n_r \geq 1$; we can therefore employ Proposition 2 and conclude that Y has solution $t = N - (n_r + n_w \cdot N) \wedge n_r \geq 1$. Substituting this solution for Y in X, using Proposition 1 to simplify the resulting equation, we find the following equivalent equation for X:

$$\begin{array}{l}(\nu X(n_r, n_w, t{:}\mathbb{N}) = ((t \geq 1 \implies (X(n_r + 1, n_w, t-1))) \\ \quad \wedge (n_r > 0 \implies X(n_r - 1, n_w, t+1)) \wedge (t \geq N \implies X(n_r, n_w + 1, t - N)) \\ \quad\quad \wedge (n_w > 0 \implies X(n_r, n_w - 1, t + N)) \wedge t = N - (n_r + n_w \cdot N)))\end{array}$$

Another application of Proposition 2, immediately leads to the solution $t = N - (n_r + n_w \cdot N)$ for X. Thus, writers cannot start writing while readers are active if initially the values for n_r, n_w, t satisfy $t = N - (n_r + n_w \cdot N)$.

Mutual exclusion can also be expressed by a single μ-calculus formula with data variables; then invariants linking process and formula variables are required.

7 Closing Remarks

Techniques and concepts for solving PBESs have been studied in detail [7]. Among these is the concept of *invariance*, which has been instrumental in solving verification problems that were studied in e.g. [7,2]. In this paper, we further studied the notion of invariance and show that the accompanying theory is impractical for PBESs in which *open* equations occur. We have proposed a stronger notion of invariance, called *global invariance*, and phrased an invariance theorem that remedies the issues of the invariance theorem of [7]. We moreover have

shown that our notion of invariance is preserved by three important solution-preserving PBES manipulations. This means that, unlike the notion of invariance of [7], global invariants can be used in combination with these manipulations when solving equation systems. As a side-result, we obtain a partial answer to an open question put forward in [7], concerning a specific pattern for PBESs.

We continued by demonstrating that invariants for processes automatically yield global invariants in the PBESs resulting from two standard verification encodings, viz. the encoding of the first-order modal μ-calculus model checking problem and the encoding of branching bisimulation for two (possibly infinite) transition systems. This means that in the PBES verification methodology, one can take advantage of established techniques for checking and discovering process invariants. We conjecture that many such techniques, see e.g. [12,13], can be put to use for (automatically) discovering global invariants in PBESs. Additional research is of course needed to substantiate this conjecture.

Acknowledgements. The authors would like to thank Jan Friso Groote for valuable feedback.

References

1. Bezem, M.A., Groote, J.F.: Invariants in process algebra with data. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 401–416. Springer, Heidelberg (1994)
2. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence checking for infinite systems using parameterized boolean equation systems. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 120–135. Springer, Heidelberg (2007)
3. van Dam, A., Ploeger, B., Willemse, T.A.C.: Instantiation for parameterised boolean equation systems. In: Proceedings of ICTAC 2008 (to appear, 2008)
4. Gallardo, M.M., Joubert, C., Merino, P.: Implementing influence analysis using parameterised boolean equation systems. In: Proceedings of ISOLA 2006. IEEE Computer Society Press, Los Alamitos (2006)
5. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
6. Groote, J.F., Willemse, T.A.C.: Model-checking processes with data. Sci. Comput. Program 56(3), 251–273 (2005)
7. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. Theor. Comput. Sci 343(3), 332–369 (2005)
8. Mader, A.: Verification of Modal Properties Using Boolean Equation Systems. PhD thesis, Technische Universität München (1997)
9. Mateescu, R.: Local model-checking of an alternation-free value-based modal mu-calculus. In: Proc. 2nd Int'l Workshop on VMCAI (September 1998)
10. Mateescu, R.: Vérification des propriétés temporelles des programmes paralléles. PhD thesis, Institut National Polytechnique de Grenoble (1998)
11. Orzan, S.M., Willemse, T.A.C.: Invariants for parameterised boolean equation systems. CS-Report 08-17, Eindhoven University of Technology (2008)

12. Pandav, S., Slind, K., Gopalakrishnan, G.: Counterexample guided invariant discovery for parameterized cache coherence verification. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 317–331. Springer, Heidelberg (2005)
13. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
14. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. Formal Methods in System Design 32(1), 25–55 (2008)
15. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Mathematics 5(2), 285–309 (1955)
16. Zhang, D., Cleaveland, R.: Fast generic model-checking for data-based systems. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 83–97. Springer, Heidelberg (2005)

Unfolding-Based Diagnosis of Systems with an Evolving Topology*

Paolo Baldan[1], Thomas Chatain[2], Stefan Haar[3], and Barbara König[4]

[1] Dipartimento di Matematica Pura e Applicata, Università di Padova, Italy
[2] LSV, ENS Cachan, CNRS, INRIA, France
[3] INRIA, France, and University of Ottawa, Canada
[4] Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, Germany

Abstract. We propose a framework for model-based diagnosis of systems with mobility and variable topologies, modelled as graph transformation systems. Generally speaking, model-based diagnosis is aimed at constructing explanations of observed faulty behaviours on the basis of a given model of the system. Since the number of possible explanations may be huge we exploit the unfolding as a compact data structure to store them, along the lines of previous work dealing with Petri net models. Given a model of a system and an observation, the explanations can be constructed by unfolding the model constrained by the observation, and then removing incomplete explanations in a pruning phase. The theory is formalised in a general categorical setting: constraining the system by the observation corresponds to taking a product in the chosen category of graph grammars, so that the correctness of the procedure can be proved by using the fact that the unfolding is a right adjoint and thus it preserves products. The theory thus should be easily applicable to a wide class of system models, including graph grammars and Petri nets.

1 Introduction

The *event-oriented model-based diagnosis* problem is a classical topic in discrete event systems [7,15]. Given an observed alarm stream, the aim is to provide *explanations* in terms of actual system behaviours. Some events of the system are observable (alarms) while others are not. In particular, fault events are usually unobservable; therefore, fault diagnosis is the main motivation of the diagnosis problem. Given a sequence (or partially ordered set) of observable events, the diagnoser has to find all possible behaviours of the model explaining the observation, thus allowing the deduction of invisible causes (faults) of visible events (alarms). The paper [16] provides a survey on fault diagnosis in this direction.

Since the number of possible explanations may be huge, especially in the case of highly concurrent systems, it is advisable to employ space-saving methods.

* Supported by the MIUR Project ART, RNRT project SWAN, INRIA Sabbatical program, the DFG project SANDS and CRUI/DAAD Vigoni.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 203–218, 2008.

In [16,10], the global diagnosis is obtained as the fusion of local decisions: this *distributed* approach allows one to factor explanations over a set of local observers and diagnoses, rather than centralizing the data storage and handling.

We will build here upon the approach of [5] where diagnoses are stored in the form of unfoldings. The unfolding of a system fully describes its concurrent behaviour in a single branching structure, representing all the possible computation steps and their mutual dependencies, as well as all reachable states; the effectiveness of the approach lies in the use of partially ordered runs, rather than interleavings, to store and handle explanations extracted from the system model.

While [5] and subsequent work in this direction was mainly directed to Petri nets, here we face the diagnosis problem in mobile and variable topologies. This requires the development of a model-based diagnosis approach which applies to other, more expressive, formalisms. Unfoldings of extensions of Petri nets where the topology may change dynamically were studied in [8,6]. Here we focus on the general and well-established formalism of graph transformation systems.

In order to retain only the behaviour of the system that matches the observations, it is not the model itself that is unfolded, but the product of the model with the observations, which represents the original system constrained by the observation; under suitable observability assumptions, a finite prefix of the unfolding is sufficient. The construction is carried out in a suitably defined category of graph grammars, where such a product can be shown to be the categorical product. A further *pruning* phase is necessary in order to remove incomplete explanations that are only valid for a prefix of the observations.

We show the correctness of this technique, i.e., we show that the runs of the unfolding properly capture all those runs of the model which explain the observation. This non-trivial result is obtained by using the fact that unfolding for graph grammars is a coreflection, hence it preserves limits (and especially products, such as the product of the model and the observation). In order to ensure that the product is really a categorical product, special care has to be taken in the definition of the category.

Additional technical details and the proofs of all the results can be found in the full version of the paper [4].

2 Graph Grammars and Grammar Morphisms

In this section we summarise the basics of graph rewriting in the *single-pushout* (SPO) approach [13]. We introduce a category of graph grammars, whose morphisms are a variation of those in [2] and we characterise the induced categorical product, which turns out to be adequate for expressing the notion of composition needed in our diagnosis framework. Then we show that the unfolding semantics smoothly extends to this setting, arguing that the unfolding construction can still be characterised categorically as a universal construction. The proof relies on the results in [2]; this motivates our choice of the SPO approach as opposed to the more classical *double-pushout* (DPO) approach, for graph rewriting.

2.1 Graph Grammars and Their Morphisms

Given a partial function $f : A \rightharpoonup B$ we write $f(a)\downarrow$ whenever f is defined on $a \in A$ and $f(a)\uparrow$ whenever it is undefined. We will denote by $dom(f)$ the *domain* of f, i.e., the set $\{a \in A \mid f(a)\downarrow\}$. Let $f, g : A \rightharpoonup B$ be two partial functions. We will write $f \leq g$ when $dom(f) \subseteq dom(g)$ and $f(x) = g(x)$ for all $x \in dom(f)$.

For a set A, we denote by A^* the set of sequences over A. Given $f : A \rightharpoonup B$, the symbol $f^* : A^* \to B^*$ denotes its extension to sequences defined by $f^*(a_1 \dots a_n) = f(a_1) \dots f(a_n)$, where it is intended that the elements on which f is undefined are "forgotten". Specifically, $f^*(a_1 \dots a_n) = \varepsilon$ whenever $f(a_i) \uparrow$ for any $i \in \{1, \dots, n\}$. Instead, $f^{\perp}: A^* \rightharpoonup B^*$ denotes the *strict* extension of f to sequences, satisfying $f^{\perp}(a_1 \dots a_n) \uparrow$ whenever $f(a_i) \uparrow$ for some $i \in \{1, \dots, n\}$.

A *(hyper)graph* G is a tuple (N_G, E_G, c_G), where N_G is a set of nodes, E_G is a set of edges and $c_G : E_G \to N_G^*$ is a connection function. Given a graph G we will write $x \in G$ to say that x is a node or edge in G, i.e., $x \in N_G \cup E_G$.

Definition 1 (partial graph morphism). *A* partial graph morphism $f : G \rightharpoonup H$ *is a pair of partial functions* $f = \langle f_N : N_G \rightharpoonup N_H, f_E : E_G \rightharpoonup E_H \rangle$ *such that:*

$$c_H \circ f_E \leq f_N^{\perp} \circ c_G \ (*)$$

We denote by **PGraph** *the category of hypergraphs and partial graph morphisms. A morphism is called* total *if both components are total, and the corresponding subcategory of* **PGraph** *is denoted by* **Graph**.

Notice that, according to condition (*), if f is defined on an edge then it must be defined on all its adjacent nodes: this ensures that the domain of f is a well-formed graph. The inequality in condition (*) ensures that *any* subgraph of a graph G can be the domain of a partial morphism $f : G \rightharpoonup H$.

We will work with *typed graphs* [9,14], which are graphs labelled over a structure that is itself a graph, called the *graph of types*.

Definition 2 (typed graph). *Given a graph T, a* typed graph *G over T is a graph $|G|$, together with a total morphism $t_G : |G| \to T$. A* partial morphism *between T-typed graphs $f : G_1 \rightharpoonup G_2$ is a partial graph morphism $f : |G_1| \rightharpoonup |G_2|$ consistent with the typing, i.e., such that $t_{G_1} \geq t_{G_2} \circ f$. A typed graph G is called* injective *if the typing morphism t_G is injective. The category of T-typed graphs and partial typed graph morphisms is denoted by T-***PGraph**.

Definition 3 (graph production, direct derivation). *Fixing a graph T of types, a* (T-typed graph) production *q is an injective partial typed graph morphism $L_q \overset{r_q}{\rightharpoonup} R_q$. It is called* consuming *if r_q is not total. The typed graphs L_q and R_q are called* left-hand side *and* right-hand side *of the production.*

Given a typed graph G and a match*, i.e., a total injective morphism $g : L_q \to G$, we say that there is a* direct derivation *δ from G to H using q (based on g), written $\delta : G \Rightarrow_q H$, if there is a pushout square in T-***PGraph** *as on the right.*

$$\begin{array}{ccc} L_q & \xrightarrow{r_q} & R_q \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle h} \\ G & \xrightarrow[d]{} & H \end{array}$$

Roughly speaking, the rewriting step removes from G the image of the items of the left-hand side which are not in the domain of r_q, namely $g(L_q - dom(r_q))$,

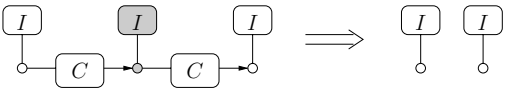


Fig. 1. Dangling edge removal in SPO rewriting

adding the items of the right-hand side which are not in the image of r_q, namely $R_q - r_q(dom(r_q))$. The items in the image of $dom(r_q)$ are "preserved" by the rewriting step (intuitively, they are accessed in a "read-only" manner). Additionally, whenever a node is removed, all the edges incident to such a node are removed as well. For instance, consider production **fail** at the bottom of Fig. 2. Its left-hand side contains a unary edge (i.e., an edge connected to only one node) and its right-hand side is the empty graph. Nodes and edges are represented as circles and boxes, respectively. The application of **fail** to a graph is illustrated in Fig. 1, where the match of the left-hand side is indicated as shaded.

Definition 4 (typed graph grammar). *A* (T-typed) SPO graph grammar $\mathcal{G}$ *is a tuple* $\langle T, G_s, P, \pi, \Lambda, \lambda \rangle$*, where* G_s *is the* (typed) start graph*,* P *is a set of* production names*,* π *is a function which associates to each name* $q \in P$ *a production* $\pi(q)$*, and* $\lambda : P \to \Lambda$ *is a labelling over the set* Λ*. A graph grammar is* consuming *if all the productions in the range of* π *are consuming.*

As standard in unfolding approaches, in the paper we will consider *consuming* graph grammars only, where each production deletes some item. Hereafter, when omitted, we will assume that the components of a given graph grammar $\mathcal{G}$ are $\langle T, G_s, P, \pi, \Lambda, \lambda \rangle$. Subscripts carry over to the component names.

For a graph grammar $\mathcal{G}$ we denote by $Elem(\mathcal{G})$ the set $N_T \cup E_T \cup P$. As a convention, for each production name q the corresponding production $\pi(q)$ will be $L_q \overset{r_q}{\rightarrowtail} R_q$. Without loss of generality, we will assume that the injective partial morphism r_q is a partial inclusion (i.e., that $r_q(x) = x$ whenever defined). Moreover we assume that the domain of r_q, which is a subgraph of both $|L_q|$ and $|R_q|$, is the *intersection* of these two graphs, i.e., that $|L_q| \cap |R_q| = dom(r_q)$, componentwise. Since in this paper we work only with typed notions, we will usually omit the qualification "typed", and, sometimes, we will not indicate explicitly the typing morphisms.

In the sequel we will often refer to the runs of a grammar defined as follows.

Definition 5 (runs of a grammar). *Let* $\mathcal{G}$ *be a graph grammar. Then* $Runs(\mathcal{G})$ *contains all sequences* $r_1 r_2 \ldots r_n$ *where* $r_i \in P$ *and* $G_s \overset{r_1}{\Rightarrow} G_1 \overset{r_2}{\Rightarrow} G_2 \ldots \overset{r_n}{\Rightarrow} G_n$*.*

Example. As a running example we will consider the graph grammar $\mathcal{M}$ whose start graph and productions are given in Fig. 2. It models a network with mobility whose nodes are either senders (labelled S), receivers (R) or intermediary nodes (I). Senders may send messages which can then cross connections and should finally arrive at a receiver. However, a connection may be spontaneously

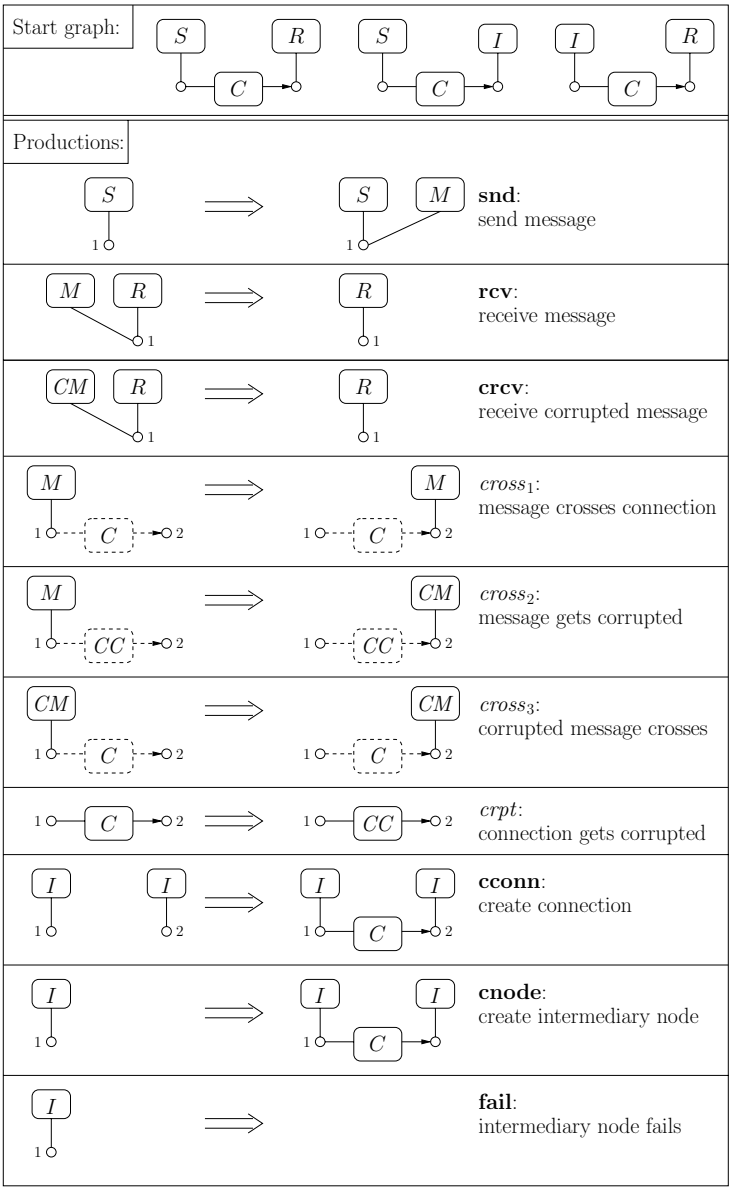


Fig. 2. Example grammar $\mathcal{M}$: message passing over possibly corrupted connections

corrupted, which causes the corruption of any message which crosses it. The network is variable and of unbounded size as we allow the creation of a new connection between existing intermediary nodes and the creation of a new connection leading from an existing intermediary node to a new intermediary node.

Productions (and the corresponding partial morphisms) are depicted as follows: edges that are deleted or created are drawn with solid lines, whereas edges that are preserved are indicated with dashed lines. Nodes which are preserved are indicated with numbers, whereas newly created nodes are not numbered.

Productions that should be observable (a notion that will be made formal in Section 4) are indicated by bold face letters.

We next define the class of grammars which will focus on in the development.

Definition 6 (semi-weighted SPO graph grammars). *A grammar $\mathcal{G}$ is* semi-weighted *if (i) the start graph G_s is injective, and (ii) for each $q \in P$, for any x, y in $|R_q| - |L_q|$ if $t_{R_q}(x) = t_{R_q}(y)$ then $x = y$, i.e., the right-hand side graph R_q is injective on the "produced part" $|R_q| - |L_q|$.*

Intuitively, conditions (i) and (ii) ensure that in a semi-weighted grammar each item generated in a computation has a uniquely determined causal history, a fact which is essential for the validity of Theorem 15.

Note that grammar $\mathcal{M}$ of Fig. 2 is not semi-weighted (if we assume the simplest type graph that contains one node and exactly one edge for every edge label). It could easily be converted into a semi-weighted grammar, for instance by creating the start graph (which is not injectively typed) step by step. However, for the sake of simplicity we do not carry out this construction in the paper.

A grammar morphism consists of a (partial) mapping between production names and a component specifying the (multi)relation between the type graphs. A morphism must "preserve" the graphs underlying corresponding productions of the two grammars as well as the start graphs. Since these conditions are exactly the same as in [2] and they are not relevant for understanding this paper, in the sequel we will refer to the morphisms in [2], making explicit only the new condition regarding the labelling. The interested reader can find the details in the full version [4].

Definition 7 (grammar morphism). *Let $\mathcal{G}_i$ ($i \in \{1, 2\}$) be graph grammars such that $\Lambda_2 \subseteq \Lambda_1$. A* grammar morphism *$f : \mathcal{G}_1 \to \mathcal{G}_2$ is a morphism in the sense of [2, Def. 15] where the component on productions, i.e., the partial function $f_P : P_1 \rightharpoonup P_2$, additionally satisfies, for all $q_1 \in P_1$*

$$f_P(q_1) \downarrow \quad \textit{iff} \quad \lambda_1(q_1) \in \Lambda_2 \textit{ and, in this case, } \lambda_2(f_P(q_1)) = \lambda_1(q_1).$$

Note that a morphism from $\mathcal{G}_1$ to $\mathcal{G}_2$ might exist only when $\Lambda_2 \subseteq \Lambda_1$.

Definition 8 (category of graph grammars). *We denote by* **GG** *the category where objects are* SPO *graph grammars and arrows are grammar morphisms. By* **SGG** *we denote the full subcategory of* **GG** *having semi-weighted graph grammars as objects.*

The choice of grammar morphisms and, in particular, the conditions on the labelling, lead to a categorical product suited for composing two grammars $\mathcal{G}_1$ and $\mathcal{G}_2$: productions with labels in $\Lambda_1 \cap \Lambda_2$ are forced to be executed in a synchronous way, while the others are executed independently in the two components.

Proposition 9 (product of graph grammars). *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two graph grammars. Their product object $\mathcal{G} = \mathcal{G}_1 \times \mathcal{G}_2$ in* **GG** *is defined as follows:*

- $T = T_1 \uplus T_2$;
- $G_s = G_{s1} \uplus G_{s2}$, *with the obvious typing;*
- $P = \{(p_1, p_2) \mid \lambda_1(p_1) = \lambda_2(p_2)\} \cup \{(p_1, \emptyset) \mid \lambda_1(p_1) \notin \Lambda_2\} \cup \{(\emptyset, p_2) \mid \lambda_2(p_2) \notin \Lambda_1\}$;
- $\pi(p_1, p_2) = \pi_1(p_1) \uplus \pi_2(p_2)$, *where $\pi_i(\emptyset)$ is the empty rule $\emptyset \to \emptyset$;*
- $\Lambda = \Lambda_1 \cup \Lambda_2$;
- $\lambda(p_1, p_2) = \lambda_i(p_i)$, *for any $i \in \{1, 2\}$ such that $p_i \neq \emptyset$;*

where, p_1 and p_2 range over P_1 and P_2, respectively, and disjoint unions are taken componentwise. If $\mathcal{G}_1, \mathcal{G}_2$ are both semi-weighted grammars, then $\mathcal{G}$ as defined above is semi-weighted, and it is the product of $\mathcal{G}_1$ and $\mathcal{G}_2$ in **SGG**.

2.2 Occurrence Grammars and Unfolding

A grammar $\mathcal{G}$ is *safe* if (i) for all H such that $G_s \Rightarrow^* H$, H is injective, and (ii) for each $q \in P$, the left- and right-hand side graphs L_q and R_q are injective.

In words, in a safe grammar each graph G reachable from the start graph is injectively typed, and thus we can identify it with the corresponding subgraph $t_G(|G|)$ of the type graph. With this identification, a production can only be applied to the subgraph of the type graph which is the image via the typing morphism of its left-hand side. Thus, according to its typing, we can think that a production *produces*, *preserves* or *consumes* items of the type graph, and using a net-like language, we speak of pre-set, context and post-set of a production, correspondingly. Intuitively the type graph T stands for the *places* of a net, whereas the productions P represent the *transitions*.

Definition 10 (pre-set, post-set and context of a production). *Let $\mathcal{G}$ be a graph grammar. For any production $q \in P$ we define its* pre-set $^\bullet q$, context $\underline{q}$ *and* post-set $q^\bullet$ *as the following subsets of $E_T \cup N_T$:*

$$^\bullet q = t_{L_q}(|L_q| - |dom(r_q)|) \quad \underline{q} = t_{L_q}(|dom(r_q)|) \quad q^\bullet = t_{R_q}(|R_q| - r_q(|dom(r_q)|)).$$

Symmetrically, for each item $x \in T$ we define $^\bullet x = \{q \in P \mid x \in q^\bullet\}$, $x^\bullet = \{q \in P \mid x \in {}^\bullet q\}$, $\underline{x} = \{q \in P \mid x \in \underline{q}\}$.

Causal dependencies between productions are captured as follows.

Definition 11 (causality). *The* causality *relation of a grammar $\mathcal{G}$ is the (least) transitive relation $<$ over $Elem(\mathcal{G})$ satisfying, for any node or edge $x \in T$, and for productions $q, q' \in P$,*

1. *if $x \in {}^\bullet q$ then $x < q$;*
2. *if $x \in q^\bullet$ then $q < x$;*
3. *if $q^\bullet \cap \underline{q'} \neq \emptyset$ then $q < q'$.*

As usual $\leq$ is the reflexive closure of $<$. Moreover, for $x \in Elem(\mathcal{G})$ we denote by $\lfloor x \rfloor$ the set of causes of x in P, namely $\{q \in P \mid q \leq x\}$.

As in Petri nets with read arcs, the fact that a production application not only consumes and produces, but also preserves a part of the state, leads to a form

of asymmetric conflict between productions; for a thorough discussion of asymmetric event structures see [1].

Definition 12 (asymmetric conflict). *The* asymmetric conflict relation *of a grammar $\mathcal{G}$ is the binary relation $\nearrow$ over the set of productions, defined by:*

1. *if $q \cap {}^\bullet q' \neq \emptyset$ then $q \nearrow q'$;*
2. *if ${}^\bullet q \cap {}^\bullet q' \neq \emptyset$ and $q \neq q'$ then $q \nearrow q'$;*
3. *if $q < q'$ then $q \nearrow q'$.*

Intuitively, whenever $q \nearrow q'$, q can never follow q' in a computation. This holds when q preserves something deleted by q' (Condition 1), trivially when q and q' are in conflict (Condition 2) and also when $q < q'$ (Condition 3). Conflicts (in acyclic grammars) are represented by cycles of asymmetric conflict: if $q_1 \nearrow q_2 \nearrow \ldots \nearrow q_n \nearrow q_1$ then $\{q_1, \ldots, q_n\}$ will never appear in the same computation.

An *occurrence grammar* is an acyclic grammar which represents, in a branching structure, several possible computations beginning from its start graph and using each production at most once. Recall that a relation $R \subseteq X \times X$ is *finitary* if for any $x \in X$, the set $\{y \in X \mid R(y, x)\}$ is finite.

Definition 13 (occurrence grammar). *An* occurrence grammar *is a safe grammar $\mathcal{O} = \langle T, G_s, P, \pi, \Lambda, \lambda \rangle$ such that*

1. *causality $<$ is irreflexive, its reflexive closure $\leq$ is a partial order, and, for any $q \in P$, the set $\lfloor q \rfloor$ is finite and asymmetric conflict $\nearrow$ is acyclic on $\lfloor q \rfloor$;*
2. *the start graph G_s is the set $Min(\mathcal{O})$ of minimal elements of $\langle Elem(\mathcal{O}), \leq \rangle$ (with the graphical structure inherited from T and typed by the inclusion);*
3. *any item x in T is created by at most one production in P, i.e., $|{}^\bullet x| \leq 1$;*

A finite occurrence grammar is deterministic *if relation $\nearrow^+$, the transitive closure of $\nearrow$, is irreflexive. We denote by* **OGG** *the full subcategory of* **GG** *with occurrence grammars as objects.*

Note that the start graph of an occurrence grammar $\mathcal{O}$ is determined by $Min(\mathcal{O})$. An occurrence grammar is deterministic if it does not contain conflicts (cycles of asymmetric conflict) so that all its productions can be executed in the same computation. In the sequel, the productions of an occurrence grammar will often be called events.

The notion of configuration captures the intuitive idea of (deterministic) computation in an occurrence grammar.

Definition 14 (configuration). *Let $\mathcal{O} = \langle T, P, \pi \rangle$ be an occurrence grammar. A* configuration *is a subset $C \subseteq P$ such that (i) for any $q \in C$ it holds $\lfloor q \rfloor \subseteq C$ and (ii) $\nearrow_C$, the asymmetric conflict restricted to C, is acyclic and finitary.*

It can be shown that, indeed, all the productions in a configuration can be applied in a derivation exactly once in any order compatible with $\nearrow$.

Since occurrence grammars are particular semi-weighted grammars, there is an inclusion functor $\mathcal{I} : \mathbf{OGG} \to \mathbf{SGG}$. Such functor has a right adjoint.

Theorem 15. *The inclusion functor $\mathcal{I} : \mathbf{OGG} \to \mathbf{SGG}$ has a right adjoint, the so-called unfolding functor $\mathcal{U} : \mathbf{SGG} \to \mathbf{OGG}$.*

As a consequence of the above result $\mathcal{U}$, as a right adjoint, preserves all limits and in particular products.

The result is a corollary of [2], which, in turn, is obtained through the explicit definition of the unfolding $\mathcal{U}(\mathcal{G})$. Given a grammar $\mathcal{G}$ the unfolding construction produces an occurrence grammar which fully describes its behaviour recording all the possible graph items which are generated and the occurrences of productions. The unfolding is obtained by starting from the start graph (as type graph), applying productions in any possible way, without deleting items but only generating new ones, and recording such production instances in the type graph. The result is an occurrence grammar $\mathcal{U}(\mathcal{G})$ and a grammar morphism $f : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$, called the *folding morphism*, which maps each item (instance of production or graph item) of the unfolding to the corresponding item of the original grammar. Because of space limitations, the construction is not formally defined here. In Section 4 we will show an example of an unfolding.

3 Interleaving Structures

Interleaving structures [3] are a semantic model which captures the behaviour of a system as the collection of its possible runs. They are used as a simpler intermediate model which helps in stating and proving the correctness of the diagnosis procedure.

An interleaving structure is essentially a collection of runs (sequences of events) satisfying suitable closure properties. Given a set E, we will denote by $E^{\odot}$ the set of sequences over E in which each element of E occurs at most once.

Definition 16 (interleaving structures). *A (labelled)* interleaving structure *is a tuple* $\mathcal{I} = (E, R, \Lambda, \lambda)$ *where* E *is a set of* events, $\lambda\colon E \to \Lambda$ *is a labelling of events and* $R \subseteq E^{\odot}$ *is the set of* runs, *satisfying: (i)* R *is prefix-closed, (ii)* R *contains the empty run* ε*, and (iii) every event* $e \in E$ *occurs in at least one run.*

The category of interleaving structures, as defined below, is adapted from [3] by changing the notion of morphisms in order to take into account the labels. This is needed to obtain a product which expresses a suitable form of synchronised composition.

Definition 17 (interleaving morphisms). *Let* $\mathcal{I}_i$ *with* $i \in \{1, 2\}$ *be interleaving structures. An* interleaving morphism *from* $\mathcal{I}_1$ *to* $\mathcal{I}_2$ *is a partial function* $\theta\colon E_1 \rightharpoonup E_2$ *on events such that*

1. $\Lambda_2 \subseteq \Lambda_1$*;*
2. *for each* $e_1 \in E_1$, $\theta(e_1)\downarrow$ *iff* $\lambda_1(e_1) \in \Lambda_2$ *and, in this case,* $\lambda_2(\theta(e_1)) = \lambda_1(e_1)$*;*
3. *for every* $r \in R_1$ *it holds that* $\theta^*(r) \in R_2$*.*

Morphism θ *is called a projection if* θ *is surjective on runs (as a function from* R_1 *to* R_2*). The category of interleaving structures and morphisms is denoted* **Ilv**.

An occurrence grammar can be easily mapped to an interleaving structure, by simply taking all the runs of the grammar.

Definition 18 (interleaving structures for occurrence grammars). *For an occurrence grammar $\mathcal{G}$ we denote by $Ilv(\mathcal{G})$ the interleaving structure which consists of all runs of $\mathcal{G}$, i.e., $Ilv(\mathcal{G}) = (P, Runs(\mathcal{G}), \Lambda, \lambda)$.*

We next characterise the categorical product in **Ilv**, which turns out to be, as in **GG**, the desired form of synchronised product.

Proposition 19 (product of interleaving structures). *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two interleaving structures. Then the product object $\mathcal{I}_1 \times \mathcal{I}_2$ is the interleaving structure $\mathcal{I} = (E, R, \Lambda, \lambda)$ defined as follows. Let*

$$E' = \{(e_1, e_2) \mid e_1 \in E_1, e_2 \in E_2, \lambda_1(e_1) = \lambda_2(e_2)\} \\ \cup \{(e_1, *) \mid e_1 \in E_1, \lambda_1(e_1) \notin \Lambda_2\} \cup \{(*, e_2) \mid e_2 \in E_2, \lambda_2(e_2) \notin \Lambda_1\}$$

and let $\pi_i : E \rightharpoonup E_i$ be the obvious (partial) projections. Then $R = \{r \in (E')^{\odot} \mid \pi_1^*(r) \in R_1, \pi_2^*(r) \in R_2\}$, $E = \{e' \in E' \mid e \text{ occurs in some run } r \in R\}$, $\Lambda = \Lambda_1 \cup \Lambda_2$ and λ is defined in the obvious way.

4 Diagnosis and Pruning

In this section we use the tools introduced so far in order to formalise the diagnosis problem. Then we show how, given a graph grammar model and an observation for such a grammar, the diagnosis can be obtained by first taking the product of the model and the observation, considering its unfolding and finally pruning such unfolding in order to remove incomplete explanations. As already mentioned, typically only a subset of the productions in the system is observable. Hence, for this section, we fix a graph grammar $\mathcal{G}$ with Λ as the set of labels, and a subset $\Lambda' \subseteq \Lambda$ of *observable labels*; an event or production is called *observable* if it has an observable label. In order to keep explanations finite, we will only consider systems that satisfy the following observability assumption (compare [15,11]): any infinite run must contain an infinite number of observable productions.

In the sequel we will need to consider the runs of a system which have a number of observable events coinciding with the number of events in the observation. For this aim the following definition will be useful.

Definition 20 (n-runs of a grammar). *Let $\mathcal{G}$ be a graph grammar. For a given $n \in \mathbb{N}$ we denote by $Runs^n(\mathcal{G})$ the set of all runs for which the number of observable productions equals n.*

The outcome of the diagnosis procedure is an occurrence grammar which, intuitively, collects all the behaviours of the grammar $\mathcal{G}$ modelling the system, which are able to "explain" the observation.

An observation can be a sequence (in the case of a single observer) or a set of sequences (in the case of multiple distributed observers) of alarms (observable events). Here we consider, more generally, partially ordered sets of observations, which can be conveniently modelled as deterministic occurrence grammars $\mathcal{O}$.

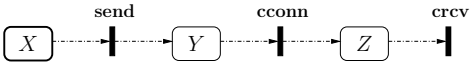


Fig. 3. A graph grammar representing an observation $\mathcal{O}$

Definition 21 (observation grammar). *An* observation grammar *for a given grammar $\mathcal{G}$, with observable labels Λ', is a (finite) deterministic occurrence grammar labelled over Λ'.*

Given a sequence of observed events, we can easily construct an observation grammar $\mathcal{O}$ having that sequence as observable behaviour. It will have a production for each event in the sequence (with the corresponding label). Each such production consumes a resource generated by the previous one in the sequence (or an initial resource in the case of the first production). The same construction applies to general partially ordered sets observations.

Example. In the example grammar $\mathcal{M}$ (see Fig. 2), assume that we have the following observation: $\mathbf{snd}_2\ \mathbf{cconn}\ \mathbf{crcv}_2$, i.e., we observe, in sequence, the sending of a message, the creation of a connection and the reception of a corrupted message. These three observations can be represented by a simple grammar $\mathcal{O}$ (see Fig. 3) with three productions, each of which either consumes an initial resource or a resource produced by the previous production. These resources are modeled as 0-ary edges (labelled X, Y, Z). The initial graph is depicted with bold lines, and the left- and right-hand sides of the productions of the occurrence grammar are indicated by a Petri-net-like notation: events are drawn with black rectangles connected to the respective edges by dashed lines.

When unfolding the product of a grammar $\mathcal{G}$ with its observation $\mathcal{O}$, we obtain a grammar $\mathcal{U} = \mathcal{U}(\mathcal{G} \times \mathcal{O})$ with a morphism $\pi\colon \mathcal{U} \to \mathcal{O}$, arising as the image through the unfolding functor of the projection $\mathcal{G} \times \mathcal{O} \to \mathcal{O}$ (since the unfolding of an occurrence grammar is the grammar itself). Now, as grammar morphisms are simulations, given the morphism $\pi\colon \mathcal{U} \to \mathcal{O}$ we know that any computation in $\mathcal{U}$ is mapped to a computation in $\mathcal{O}$. Say that a computation in $\mathcal{U}$ is a full explanation of $\mathcal{O}$ if it is mapped to a computation of O including all its productions. As $\mathcal{U}$ can still contain events belonging only to incomplete explanations, the aim of *pruning* is to remove such events.

Definition 22 (pruning). *Let $\pi\colon \mathcal{U} \to \mathcal{O}$ be a grammar morphism from an occurrence grammar $\mathcal{U}$ to an observation $\mathcal{O}$. We define the pruning of π, denoted by $Pr(\pi)$, to be the grammar obtained from $\mathcal{U}$ by removing all events (including their consequences) not belonging to the following set:*

$$\{q \in \mathcal{P}_{\mathcal{U}} \mid \exists C \in \mathit{Conf}(\mathcal{U})\colon (q \in C \wedge \pi(C) = \mathcal{P}_{\mathcal{O}})\}$$

Discussing the efficiency of pruning algorithms is outside the scope of the paper; for sequential observations an on-the-fly algorithm is discussed in [5].

As described above, the diagnosis is constructed by first taking the product of $\mathcal{G}$ with the observation (this intuitively represents the system constrained by

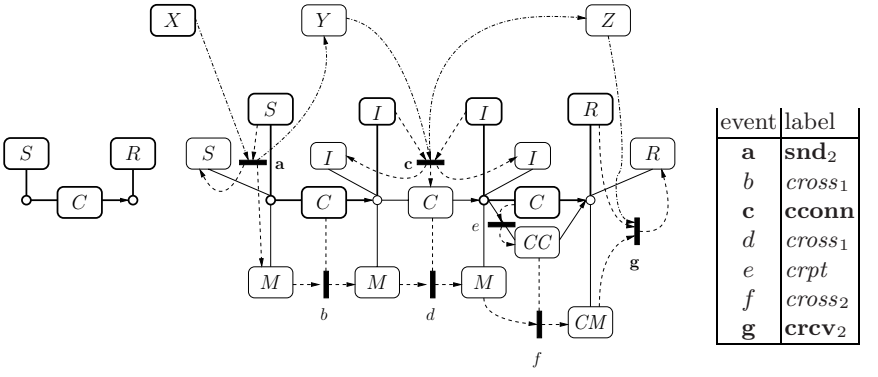


Fig. 4. Running example: prefix of the unfolding of the product

the observation). This product is then unfolded to get an explicit representation of the possible behaviours explaining the observation. Finally, a pruning phase removes from the resulting occurrence grammar the events belonging (only) to incomplete explanations. This is formalised in the definition below.

Definition 23 (diagnosis grammar). *Let $\mathcal{G}$ be the grammar modelling the system of interest and let $\mathcal{O}$ be an observation. Take the product $\mathcal{G} \times \mathcal{O}$, the right projection $\varphi : \mathcal{G} \times \mathcal{O} \to \mathcal{O}$ and consider $\pi = \mathcal{U}(\varphi) : \mathcal{U}(\mathcal{G} \times \mathcal{O}) \to \mathcal{O}$.*

Then the occurrence grammar $Pr(\pi)$ is called the diagnosis grammar *of the model and the observation and denoted by $D(\mathcal{G}, \mathcal{O})$.*

Note that since the observability assumption holds, it can easily be shown that the diagnosis grammar is finite, whenever the observation is finite.

Example. We can compute the product of grammars $\mathcal{M}$ and $\mathcal{O}$ and unfold it. For reasons of space Fig. 4 shows only a prefix of the unfolding that depicts one possible explanation: here the message is sent (event **a**) and crosses the first connection (b). Possibly concurrently a new connection between the two intermediate nodes is created (**c**), which is then also crossed by the message (d). Again in a possibly concurrent step the last connection is corrupted (e), leading to the corruption of the message (f) and its reception by the receiver (**g**). Observable events are indicated by bold face letters.

Several events of the unfolding have been left out due to space constraints, for instance:

- Events belonging to alternative explanations: the corruption of the first connection or the corruption of the newly created middle connection (or the corruption of any non-empty subset of these connections). Alternatively it might also have been the case that the other sender/receiver pair handles the message, while the connection (which is not involved in any way) is created between the two intermediate nodes.
- Events that happen concurrently but are not directly related to the failure, such as the corruption of a connection over which no message is sent.

Furthermore there are events belonging to prefixes of the unfolding that cannot be extended to a full observation. For instance, the unfolding would contain concurrent events representing sending by the right-hand sender and the creation of a new connection leading from right to left instead of left to right. However, this is a false trail since this would never cause the reception of a message by the right-hand receiver. These incomplete explanations are removed from the unfolding in the pruning phase.

Note that—due to the presence of concurrent events—the unfolding is a much more compact representation of everything that might have happened in the system than the set of all possible interleavings of events.

5 Correctness of the Diagnosis

We now show our main result, stating that the runs of the diagnosis grammar properly capture all those runs of the system model which explain the observation. This is done by exploiting the coreflection result (Theorem 15) and by additionally taking care of the pruning phase (Definition 22). We first need some technical results.

Lemma 24. *Let $\mathcal{G}_1, \mathcal{G}_2$ be two occurrence grammars. Consider the product of the two grammars and its image through the Ilv functor as shown below. Furthermore consider the product of the interleaving structures $Ilv(\mathcal{G}_1)$, $Ilv(\mathcal{G}_2)$. Then the mediating morphism δ is a projection which is total on events.*

$$\begin{array}{ccccc} Ilv(\mathcal{G}_1) & \xleftarrow{\delta_1} & Ilv(\mathcal{G}_1) \times Ilv(\mathcal{G}_2) & \xrightarrow{\delta_2} & Ilv(\mathcal{G}_2) \\ & \nwarrow^{\pi_1} & \uparrow \delta & \nearrow_{\pi_2} & \\ & & Ilv(\mathcal{G}_1 \times \mathcal{G}_2) & & \end{array}$$

To lighten the notation, hereafter, given an interleaving structure $\mathcal{I}$ we write $\lambda^*(I)$ for $\lambda^*(R_I)$. Recall that, given $f : \Lambda_1 \rightharpoonup \Lambda_2$, $f^* : \Lambda_1^* \to \Lambda_2^*$ denotes the (non-strict) extension of f to sequences. Then $f^{-1} : \mathcal{P}(\Lambda_2^*) \to \mathcal{P}(\Lambda_1^*)$ is its inverse.

Lemma 25. *Let $\mathcal{G}_1, \mathcal{G}_2$ be two occurrence grammars and let $f_i \colon \Lambda_1 \cup \Lambda_2 \rightharpoonup \Lambda_i$ ($i \in \{1, 2\}$) be the obvious partial inclusions. Then it holds that*

$$\lambda^*(Ilv(\mathcal{G}_1 \times \mathcal{G}_2)) = f_1^{-1}(\lambda_1^*(Ilv(\mathcal{G}_1))) \cap f_2^{-1}(\lambda_2^*(Ilv(\mathcal{G}_2))).$$

The next proposition shows that considering the product of the original grammar $\mathcal{G}$ and of the observation $\mathcal{O}$, taking its unfolding and the corresponding labelled runs, we obtain exactly the runs of $\mathcal{G}$ compatible with the observation.

Proposition 26. *Let $\mathcal{G}$ be a grammar and $\mathcal{O}$ an observation, where Λ is the set of labels of $\mathcal{G}$ and $\Lambda' \subseteq \Lambda$ the set of labels of $\mathcal{O}$. Furthermore let $f \colon \Lambda \rightharpoonup \Lambda'$ be the obvious partial inclusion. Then it holds that:*

$$\lambda^*(Ilv(\mathcal{U}(\mathcal{G} \times \mathcal{O}))) = \lambda^*(Runs(\mathcal{G})) \cap f^{-1}(\lambda^*(Runs(\mathcal{O}))).$$

We can conclude that the described diagnosis procedure is complete, i.e., given an observation of size n, the runs of the diagnosis grammar $D(\mathcal{G}, \mathcal{O})$ with n observable events are in 1-1 correspondence with those runs of $\mathcal{G}$ that provide

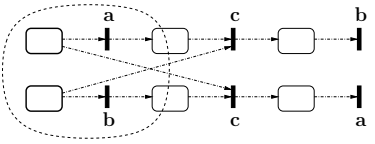


Fig. 5. Spurious runs in a diagnosis grammar

a full explanation of the observation. As a preliminary result, on the basis of Proposition 26 one could have shown that the same holds replacing the diagnosis grammar with $\mathcal{U}(\mathcal{G} \times \mathcal{O})$, i.e., the unpruned unfolding. The result below additionally shows that no valid explanation is lost during the pruning phase.

Theorem 27 (correctness of the diagnosis). *With the notation of Proposition 26 it holds that:*

$$\lambda^*(Runs^n(D(\mathcal{G}, \mathcal{O}))) = \lambda^*(Runs(\mathcal{G})) \cap f^{-1}(\lambda^*(Runs^n(\mathcal{O}))).$$

That is, the maximal interleavings of the diagnosis grammar (seen as label sequences) are exactly the runs of the model which explain the full observation.

Observe that, due to the nondeterministic nature of the diagnosis grammar, events which are kept in the pruning phase as they are part of some full explanation of the observation, can also occur in a different configuration. As a consequence, although all inessential events have been removed, the diagnosis grammar can still contain spurious configurations which cannot be extended to full explanations. As an example, consider the graph grammar $\mathcal{G}$ in Fig. 5, given in a Petri-net-like notation. Assume we observe three unordered events **a**, **b**, **c**. Then the unfolding of the product basically corresponds to $\mathcal{G}$ itself. In the pruning phase nothing is removed. However there is a configuration (indicated by the dashed closed line) that cannot be further extended to an explanation.

6 Conclusion

In this paper we formalised event-based diagnosis for systems with variable topologies, modelled as graph transformation systems. In particular we have shown how to exploit the coreflection result for the unfolding of graph grammars in order to show the correctness of a diagnosis procedure generating partially ordered explanations for a given observation.

We are confident that the approach presented in the paper, although developed for transformation systems over hypergraphs, can be generalised to the more abstract setting of adhesive categories. In particular we are currently working on a generalization of the unfolding procedure that works for SPO-rewriting in (suitable variations of) adhesive categories [12]. This would allow one to have a kind of parametric framework which can be used to instantiate the results of this paper to more general rewriting theories.

We are also interested in distributed diagnosis where every observer separately computes possible explanations of local observations that however have to be synchronized. In [3] we already considered distributed unfolding of Petri nets; for *diagnosis* however, the non-trivial interaction of distribution and pruning has to be taken into account. Distribution will require the use of pullbacks of graph morphisms, in addition to products.

References

1. Baldan, P., Corradini, A., Montanari, U.: Contextual Petri nets, asymmetric event structures and processes. Information and Computation 171(1), 1–49 (2001)
2. Baldan, P., Corradini, A., Montanari, U., Ribeiro, L.: Unfolding semantics of graph transformation. Information and Computation 205, 733–782 (2007)
3. Baldan, P., Haar, S., König, B.: Distributed unfolding of Petri nets. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 126–141. Springer, Heidelberg (2006)
4. Baldan, P., Chatain, T., Haar, S., König, B.: Unfolding-based diagnosis of systems with an evolving topology. Technical Report 2008-2, Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen (2008)
5. Benveniste, A., Fabre, E., Haar, S., Jard, C.: Diagnosis of asynchronous discrete event systems, a net unfolding approach. IEEE Transactions on Automatic Control 48(5), 714–727 (2003)
6. Bruni, R., Melgratti, H.C.: Non-sequential behaviour of dynamic nets. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 105–124. Springer, Heidelberg (2006)
7. Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Kluwer Academic Publishers, Dordrecht (1999)
8. Chatain, T., Jard, C.: Models for the supervision of web services orchestration with dynamic changes. In: AICT/SAPIR/ELETE, pp. 446–451. IEEE, Los Alamitos (2005)
9. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae 26, 241–265 (1996)
10. Fabre, E., Benveniste, A., Haar, S., Jard, C.: Distributed monitoring of concurrent and asynchronous systems. Journal of Discrete Event Dynamic Systems 15(1), 33–84 (2005)
11. Haar, S., Benveniste, A., Fabre, E., Jard, C.: Partial order diagnosability of discrete event systems using Petri net unfoldings. In: Proc. 42nd IEEE Conf. on Decision and Control (CDC) (2003)
12. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. RAIRO – Theoretical Informatics and Applications 39(3) (2005)
13. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science 109, 181–224 (1993)
14. Löwe, M., Korff, M., Wagner, A.: An Algebraic Framework for the Transformation of Attributed Graphs. In: Sleep, M.R., Plasmeijer, M.J., van Eekelen, M.C. (eds.) Term Graph Rewriting: Theory and Practice, pp. 185–199. Wiley, Chichester (1993)
15. Sampath, M., Sengupta, R., Sinnamohideen, K., Lafortune, S., Teneketzis, D.: Failure diagnosis using discrete event models. IEEE Trans. on Systems Technology 4(2), 105–124 (1996)
16. Wang, Y., Yoo, T.-S., Lafortune, S.: Diagnosis of discrete event systems using decentralized architectures. Discrete Event Dynamic Systems 17(2), 233–263 (2007)

On the Construction of Sorted Reactive Systems*

Lars Birkedal, Søren Debois, and Thomas Hildebrandt

Programming, Logic and Semantics Group
IT University of Copenhagen
{birkedal,debois,hilde}@itu.dk

Abstract. We develop a theory of *sorted bigraphical reactive systems*.

Every application of bigraphs in the literature has required an extension, a sorting, of pure bigraphs. In turn, every such application has required a redevelopment of the theory of pure bigraphical reactive systems for the sorting at hand. Here we present a general construction of sortings. The constructed sortings always sustain the behavioural theory of pure bigraphs (in a precise sense), thus obviating the need to redevelop that theory for each new application. As an example, we recover Milner's local bigraphs as a sorting on pure bigraphs.

Technically, we give our construction for ordinary reactive systems, then lift it to bigraphical reactive systems. As such, we give also a construction of sortings for ordinary reactive systems. This construction is an improvement over previous attempts in that it produces smaller and much more natural sortings, as witnessed by our recovery of local bigraphs as a sorting.

1 Introduction

Bigraphical reactive systems is a framework proposed by Milner and others [1,2,3,4] as a unifying theory of process models and as a tool for reasoning about ubiquitous computing. For process models, it has been shown that Petri-nets [5], CCS [1], various π-calculi [4,6,7], the fusion calculus [8], mobile ambients [6], and Homer [9] can all be understood as bigraphical reactive systems (although transition semantics are usually captured only approximately). Moreover, Milner recently used bigraphs as a vehicle for studying confluence, using the λ-calculus as an example [3,10]. For ubiquitous computing, bigraphical models were investigated in [11].

A bigraphical reactive system consists of a category of bigraphs and a reaction relation on those bigraphs; we can think of the bigraphs as terms modulo structural congruence and the reaction relation as term rewrite rules. The benefit of working within bigraphical reactive systems comes from their rich behavioural

* This work is funded in part by the Danish Research Agency (grants no.: 2059-03-0031 and no.: 274-06-0415) and the IT University of Copenhagen (the LaCoMoCo, BPL and CosmoBiz projects). Authors are listed alphabetically.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 218–232, 2008.

theory [1,4,2,6,12,13,14] which (a) induces a labelled transition system automatically for any reaction relation and (b) guarantees that bisimulation on that transition system is a congruence.

A category of bigraphs is formed according to a single-sorted signature, which defines the kinds of nodes found in bigraphs of that category. Single-sorted signatures are usually insufficient when we define programming languages or algebraic models: We need to constrain the combination of operators, so we need richer notions of sorting. Indeed, *every one* of [1,2,3,4,5,6,7,9,10,11] construct a richer sorting to fit the framework of bigraphs to the problem at hand.

Alas, the behavioural theory of bigraphs applies only to single-sorted (or pure) bigraphs, not to arbitrarily-sorted extensions. Hence, *also* every one of [1,2,3,4,5,6,7,9,10,11] must re-develop substantial parts of the behavioural theory. Worse, although some sortings are easy to construct [1,2,6], others require either hard work [11] or ingenuity [4,3,10] to achieve conceptually simple effects.

Up until now, it has been an open question what kinds of extensions would admit such a redevelopment. In this paper we provide a large class of extensions for which such a redevelopment is possible. Moreover, we give a method for automatically constructing sortings for such extensions.

The key observation is that most sortings in the literature exists solely to get rid of bigraphs that are meaningless for the application at hand. That is, most sortings exists solely to impose a predicate on the morphisms in the category of pure bigraphs. We give a method to automatically construct a well-behaved sorting for any decomposable such predicate. Here, a predicate P is decomposable iff it is true at every identity and $P(g \circ f)$ implies $P(g)$ and $P(f)$; all but one of the above-mentioned applications fall into this class. In particular, we prove that Milner's local bigraphs [10] arise as a sorting of pure bigraphs.

Thus, by identifying a large class of predicates for which we can construct well-behaved sortings, we make it easier to work with bigraphical reactive systems and we push back the limit for what we can hope to achieve with them.

Overview of the technical development. We ask and answer the following two questions:

1. Which sortings sustain the behavioural theory of bigraphs?
2. How do we construct such a sorting for a given problem domain?

We answer Question 1 by giving a sufficient condition for a sorting of a reactive system to sustain the behavioural theory of the well-sorted parts of the original system. We then lift both this result and previous work on sortings for reactive systems [15,6] to the present setting of bigraphical reactive systems. We answer Question 2 by giving a new family of sortings, closure sortings, all of which sustain the behavioural theory. In particular, we show how Milner's local bigraphs [10,3] arise as a special case of the closure sorting.

In more detail: Question 1. Jensen [6] found a sufficient condition, safety, for a small class of sortings for bigraphs to preserve congruence properties. In [15] we moved that condition to general sortings of reactive systems.

In the present paper we complement that result with a sufficient condition for a sorting to preserve and reflect reaction and transition semantics for any well-sort of reactive system. We say that sorting with that property "has semantic correspondence." Altogether, in the setting of reactive systems we now have sufficient conditions for a sorting to reflect congruence properties and operational semantics; this is what we mean by "sustaining the behavioural theory". We then proceed to lift these conditions to bigraphical reactive systems which, despite the name, are not an instance of ordinary reactive systems. Moreover, we argue that in general, to construct a well-behaved sorting of a bigraphical reactive system, it is sufficient to construct a well-behaved sorting of the underlying reactive system.

In more detail: Question 2. As part of the safety condition mentioned above, it is required that if a decomposable context has a sorting, then the sorting can be decomposed correspondingly. In particular, the "has a sorting" predicate P on the pure category is decomposable, i.e., $P(f \circ g)$ implies $P(f)$ and $P(g)$. Thinking in terms of sorted algebra or programming languages this is a very natural condition — a refinement of a sorting should not constrain the way a well-sorted term can be decomposed.

In [15] we discovered that banning bigraphs containing particular "bad" sub-bigraphs corresponds exactly to giving a decomposable predicate. This insight gave rise to an answer to Question 2, albeit only for reactive systems: We gave, for any predicate P on the morphisms of a category, a sorting called the predicate sorting. The predicate sorting sustains the theory of reactive systems.

In practice, however, the predicate sorting turns out to provide far more sorts than did the sortings found in an ad hoc way for the applications in [1,2,3,4,5,6,7,9,10,11]. In the present paper, we construct a new family of sortings, the closure sortings. Like the predicate sortings, closure sortings sustain the behavioural theory of both bigraphs and reactive systems. Unlike the predicate sortings, closure sortings give sorts much closer to what we find in the literature. As a (spectacular!) example, we show that Milner's Local bigraphs [10,3] are recovered in a closure sorting, by taking Milner's scoping condition as a predicate on bigraphs.

Outline. In Section 2 we revisit Leifer and Milner's classic Reactive Systems [12,13], a precursor to bigraphs. In Section 3, we recall the definition of a sorting of a reactive system and recall our previous generalisation of Jensen's safety condition. In Section 4, we give a general condition for a sorting of a reactive system to preserve dynamics up to a predicate (Theorem 1), partially answering Question 1. In Section 5, we give the closure sorting (Definition 12), partially answering Question 2 above (Theorem 2). In Section 6, we remark on lifting these partial answers to the setting of bigraphical reactive systems, thus arriving at complete answers to both questions. Finally, in Section 7, as an extended example, we recover local bigraphs as a full sub-sorting of a closure sorting (Theorem 3).

For want of space, proofs have been omitted from this extended abstract; they can be found in [16].

2 Reactive Systems

In this section, we recall Milner and Leifer's *Reactive Systems* [17,13,12]. These systems form the conceptual basis of bigraphical reactive systems. Except for the running example, this section contains no original work.

Let C be a category, and let $\mathcal{R}_\epsilon$ be an object of C. We think of morphisms with domain $\mathcal{R}_\epsilon$ as *agents* or *processes* and all other morphisms as *contexts*. A *reaction rule* (l, r) is a cospan of agents with common domain $\mathcal{R}_\epsilon$; intuitively, l and r are the left- and right-hand sides of a rewrite rule. A set $\mathcal{R}$ of reaction rules induces a *reaction relation*, $\longrightarrow$, obtained by closing reaction rules under contexts:

$$a \longrightarrow b \text{ iff } \exists f \in C, \exists (l, r) \in \mathcal{R}.\ a = f \circ l, b = f \circ r. \tag{1}$$

Altogether, these components constitute a reactive system.

Definition 1 (Reactive system). *A* reactive system *over a category C comprises a distinguished object $\mathcal{R}_\epsilon$ and a set $\mathcal{R}$ of reaction rules; the reaction rules give rise to a* reaction relation *by (1) above. We identify a reactive system with its reaction rules, writing $\mathcal{R}$ for both.*

In this definition we have omitted the notion of activity usually associated with reactive systems. Activity can be recovered as a sorting both in the case of reactive systems [16] and in the case of bigraphical reactive systems [6].

Example 1. Here is a small process language.

$$P, Q ::= 0 \mid \mathsf{a} \mid \mathsf{s} \mid (P|Q) \tag{2}$$

These are the nil process 0, atomic processes a and s, and parallel composition of processes. As usual, we consider processes up to a structural congruence comprising the commutative monoid laws for $|$ and 0. Clearly, the set of processes is isomorphic to the free, commutative monoid over $\{\mathsf{a}, \mathsf{s}\}$; that is, a category with a single object, terms up to structural congruence as morphisms, composition $f \circ g = f|g$, and identity 0. (In this case, there is no distinction between agents and contexts: all morphisms are both.) Call this category $\mathcal{C}$. We intend a to model a normal process and s to model two processes in a synchronized state. Here are the reaction rules:

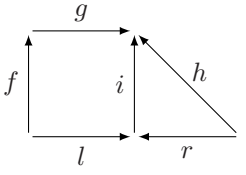
$$(\mathsf{a}|\mathsf{a},\ \mathsf{s}\ ) \qquad \text{“two processes synchronize,”} \tag{3a}$$

$$(\mathsf{s}\ ,\ \mathsf{a}|\mathsf{a}) \qquad \text{“processes drop synchronization.”} \tag{3b}$$

Here are two reactions, using first rule 3a, then 3b: $\mathsf{a}|\mathsf{a}|\mathsf{s} \longrightarrow \mathsf{s}|\mathsf{s} \longrightarrow \mathsf{s}|\mathsf{a}|\mathsf{a}$.

Leifer and Milner give a method for *deriving* labeled transitions for any reactive system. If the underlying category has sufficient relative pushouts (RPOs), then the bisimulation on those labeled transitions is a congruence. To construct labeled transitions, we take as labels minimal contexts enabling reaction. The notion of idem-pushout (IPOs) captures minimality[1].

Definition 2. *For a reactive system $\mathcal{R}$ over a category C, we define the* transition relation *by $f \xrightarrow{g} h$ iff there exists a context i and a reaction rule $(l, r) \in \mathcal{R}$ s.t. the following diagram commutes, and the square is an IPO.*



(4)

Example 2. $\mathcal{C}$ is isomorphic to the category of multisets over $\{\mathsf{a}, \mathsf{s}\}$ (with multiset union as composition). Thus $\mathcal{C}$ has pushouts, given by multiset subtraction, and thus RPOs. The pushout of multisets simply adds what is missing: The pushout of a and $\mathsf{a}|\mathsf{a}$ is a and 0, the pushout of a and s is s and a. Because IPOs are precisely the pushouts in this category, we find transitions for an agent a by taking the pushout of a and either left-hand side of the two rules.

$$\mathsf{a} \xrightarrow{\mathsf{a}} \mathsf{s} \qquad \text{by rule (3a)} \tag{5}$$

$$\mathsf{a} \xrightarrow{\mathsf{s}} \mathsf{a}|\mathsf{a}|\mathsf{a} \qquad \text{by rule (3b)} \tag{6}$$

There are no transitions from a with label $\mathsf{a}|\mathsf{a}$. A label can only add what is missing, and the agent a is only one "a" short of the left-hand side $\mathsf{a}|\mathsf{a}$ of rule (3a).

As mentioned, the bisimulation on such derived transition systems is a congruence whenever the underlying category has RPOs [13,12].

Proposition 1 ([12]). *Let $\mathcal{R}$ be a reactive system on a category C. If C has RPOs, then the bisimulation on the derived transitions is a congruence.*

3 Sortings

In this section, we recall the notion of *sorting* [15] for categories. As in the previous section, this section contains nothing new but the running example.

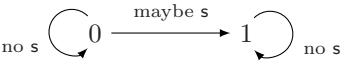
Definition 3 (Sorting). *A* sorting *of a category C is a functor $F : X \to C$ that is faithful and surjective on objects.*

We shall consistently confuse a sorting functor F with its domain: We write $F \to C$, and we speak interchangeably of F as a category and as a functor.

[1] For brevity, we omit the definitions of both RPOs and IPOs.

Example 3. Suppose we want to restrict our process language such that at most two processes are synchronized at any time, i.e., no process can contain a subterm $\mathsf{s}|\mathsf{s}$. We cannot just stipulate that there are no such terms, because the composition of s and s would then be undefined. Instead, we define a sorting $\mathcal{F} \to \mathcal{C}$ that refines the single homset of $\mathcal{C}$ into three.

The category $\mathcal{F}$ has two objects, 0 and 1. The homsets $\mathcal{F}(0,0)$ and $\mathcal{F}(1,1)$ are identical, comprising morphisms of $\mathcal{C}$ that *do not* contain an s; $\mathcal{F}(0,1)$ comprises morphisms with *at most one* s. Here is a sketch of $\mathcal{F}$.



Composition is defined as in $\mathcal{C}$; it is easy to check that this composition is well-defined on our refined homsets.

Usually when we construct a sorting $F \to C$, we will want to apply Proposition 1 to the category F. Hence, we want sortings that allow us to infer the existence of RPOs in F from the existence of RPOs in C. The following notion of *transfer* helps us do that.

Definition 4 (Transfer of RPOs). *A sorting $F \to C$* transfers RPOs *iff whenever the image of a square s in F has an RPO, then that RPO has an F-preimage that is an RPO for s.*

Jensen gives a sufficient condition, *safety*, for a sorting to transfer RPOs [6]; we generalized that condition in [15] (see also [16]; again for brevity, we omit the definition here).

Proposition 2 ([6,15]). *Let $F \to C$ be a safe sorting. Then F transfers RPOs, and, if C has RPOs then so does F.*

4 Semantic Correspondence

In Example 3 above we used the sorting $H \to C$ to get rid of morphisms not satisfying the predicate "contain at most one s". Thus, that sorting is a *realization of a predicate* on the morphisms of C. Of the sortings in [1,2,3,4,5,6,7,9,10,11], only the one in [7] is not a realization of a predicate.

However, not every sorting realizing a predicate is equally interesting; we must require also that the sorted category supports the same reactive systems as the original one, at least when we restrict our attention to the "good" morphisms of the original category. This semantic correspondence is part of what we mean by "sustaining the behavioural theory". In this section, towards answering Question 1, we give a sufficient condition for a sorting of a reactive system to admit such semantic correspondence.

This result generalizes our previous [15], where we proved that a particular sorting has semantic correspondence[2]. As in that paper, we will consider only

[2] Although in that paper, we used the somewhat inaccurate term "preserves semantics" rather than the present "has semantic correspondence".

predicates P that are decomposable, that is, that are true at every identity and have $P(f \circ g)$ implies $P(f)$ and $P(g)$. This restriction is not very severe; for free structures, decomposable predicates are precisely those that prohibit terms containing some given set of subterms [15].

To formalize the notion of "semantic correspondence", we need the notion of reflection of a reactive system.

Definition 5 (Reflection of a reactive system). *Let $F \to C$ be a sorting, and let $\mathcal{R}$ be a reactive system on C. For a preimage $\hat{\mathcal{R}}_\epsilon$ of $\mathcal{R}_\epsilon$, the* reflection of $\mathcal{R}$ at $\hat{\mathcal{R}}_\epsilon$ *is $\hat{\mathcal{R}}$, where*

$$\hat{\mathcal{R}} = \{(f, g) \mid \exists x. f, g : \hat{\mathcal{R}}_\epsilon \to x \wedge (F(f), F(g)) \in \mathcal{R}\}. \tag{7}$$

We can now define semantic correspondence precisely.

Definition 6 (Correspondence of reactions, transitions). *Let $F \to C$ be a sorting, let $\mathcal{R}, \mathcal{S}$ be reactive systems on F, C, respectively, and let P be a decomposable predicate on C. The sorting $F \to C$ has* correspondence of P-transitions for $\mathcal{R}, \mathcal{S}$ *iff whenever f, g, h are morphisms of C with $P(g \circ f)$ and $P(h)$, then*

$$\begin{aligned} f \xrightarrow{\ g\ } h \quad \textit{iff} \quad & \exists \hat{f}, \hat{g}, \hat{h}.\ \hat{f} \xrightarrow{\ \hat{g}\ } \hat{h} \textit{ where} \\ & F(\hat{f}) = f, F(\hat{g}) = g, F(\hat{h}) = h. \end{aligned} \tag{8}$$

We define correspondence of P-reactions similarly.

Note that despite F faithful, $\hat{f}, \hat{g}, \hat{h}$ are not necessarily unique.

Before we can define the general notion of semantic correspondence, we need first a notion of a reactive system respecting a predicate.

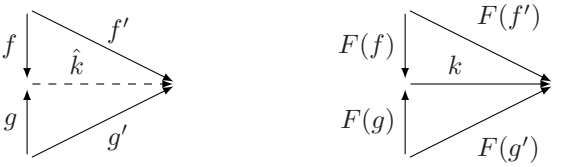
Definition 7 (P-respecting reactive system). *Let $\mathcal{R}$ be a reactive system and let P be a predicate on $\mathcal{R}$'s underlying category. We say that $\mathcal{R}$ is a P-*respecting reactive system *iff every rule $(l, r) \in \mathcal{R}$ has both $P(l)$ and $P(r)$.*

We lift the correspondence from a property of reactive systems to a property of sortings: A sorting $F \to C$ has semantic correspondence iff any P-respecting reactive system on C has a reflection in F with which it is in semantic correspondence.

Definition 8 (Semantic correspondence). *We say that a sorting $F \to C$ has* semantic correspondence up to P *iff for any P-respecting reactive system $\mathcal{R}$ on C, there exists a reflection $\hat{\mathcal{R}}$ of $\mathcal{R}$ at some $\hat{\mathcal{R}}_\epsilon$ such that F has correspondence of P-reactions and P-transitions for $\hat{\mathcal{R}}, \mathcal{R}$.*

Theorem 1 below gives a sufficient condition for a sorting to have semantic correspondence. This condition depends on the notion of "weak joint opfibration", which was introduced in [15]. Intuitively, a weak joint opfibration is a functor which has most general lifts of every cospan in its domain. The following notion of "jointly opcartesian" captures such most general lifts.

Definition 9 (Jointly opcartesian). *Let $F \to C$ be a functor. A cospan f, g in C is said to be* jointly opcartesian *iff whenever f', g' is a cospan, f, f' is a span, and g, g' is a span (see the diagram below, left side) with $F(f') = k \circ F(f)$ and $F(g') = k \circ F(g)$ (see the diagram below, right side), then there exists a unique lift $\hat{k}$ of k s.t. $f' = \hat{k} \circ f$ and $g' = \hat{k} \circ f'$.*



Example 4. In the sorting $\mathcal{F} \to \mathcal{C}$ of Example 3, the cospan a, a has a jointly opcartesian lift $\mathsf{a} : 0 \longrightarrow 0 \longleftarrow 0 : \mathsf{a}$; the cospan s, a has a jointly opcartesian lift $\mathsf{s} : 0 \longrightarrow 1 \longleftarrow 0 : \mathsf{a}$.

Armed with jointly opcartesian lifts, we define weak joint opfibrations.

Definition 10 (Weak joint opfibration [15]). *A functor $F : X \to C$ is a* weak joint opfibration *iff whenever $F(f), F(g)$ form a cospan in C, then there exists a jointly opcartesian pair $\hat{f}, \hat{g}$ with $F(\hat{f}) = F(f)$ and $F(\hat{g}) = F(g)$.*

Finally, getting back to our sufficient condition for a sorting to have semantic respondents, we now need only the following auxiliary definition.

Definition 11. *Let $F \to C$ be a sorting, let x be an object of F, and let P be a predicate on C. We say F* reflects P at x *if every morphism $f : F(x) \to c$ with $P(f)$ has a lift at x.*

Theorem 1. *Let $F \to C$ be a sorting, let P be a decomposable predicate on C, let $\mathcal{R}$ be a reactive system on C, and let $\hat{\mathcal{R}}_\epsilon$ be an F-preimage of $\mathcal{R}_\epsilon$. Then (a) the reactions of the reflection of $\mathcal{R}$ at $\hat{\mathcal{R}}_\epsilon$ correspond to the P-reactions of $\mathcal{R}$ if (i) the image of F is P, (ii) F reflects P at $\hat{\mathcal{R}}_\epsilon$, and (iii) F is a weak joint opfibration. Moreover, (b) the transitions of the reflection of $\mathcal{R}$ at $\hat{\mathcal{R}}_\epsilon$ corresponds to the P-transitions of $\mathcal{R}$ if also F transfers and preserves RPOs. In this case, F has semantic correspondence up to P.*

Between them, Theorem 1 above and Proposition 2 recalled in the preceding section answer Question 1. The former gives us a sufficient condition for a sorting to admits the necessary operational semantics; the latter gives as a sufficient condition for a sorting to reflect congruence properties. Technically, this answer applies only to sortings a reactive systems, however, it is straightforward to lift the answer to the case of bigraphical reactive systems; more on such in Section 6.

5 Closure Sortings

In this section, we answer Question 2: We define, for each decomposable predicate P, a *Closure sorting* realizing P. Every closure sorting transfers RPOs and

has semantic correspondence up to P. We define closure sortings in terms of categories and reactive systems here, but in Section 6, we remark on lifting them to bigraphs.

Suppose we want to construct a sorting realizing a decomposable predicate P on the morphisms of a category C. The basic problem here is that we may have morphisms $f : x \to y$ and $g : y \to z$ which satisfy P individually but not when composed, i.e., $P(f)$ and $P(g)$, but $\neg P(g \circ f)$. At each preimage of y, we must choose whether to admit f or g. We make this choice explicit in the closure sorting by taking as pre-images for an object y pairs (F, G) of sets of morphisms such that every $g \in G$ has domain y and can be composed with every $f \in F$, that is, f has codomain y and $P(g \circ f)$. This approach leads to too many objects in the sorted category, so we further insist that (F, G) be *maximal*, that is, that adding morphisms to either F or G would violate $f \in F, g \in G \implies P(g \circ f)$.

To formalize maximality, first define $g \perp f$ iff $P(g \circ f)$. Then define, for any $c \in C$, operators $\triangle$ and $\triangledown$ by

$$\begin{aligned} \triangle F &= \{g : c \to x \mid g \perp F, \text{ any } x \in C\} \\ \triangledown G &= \{f : y \to c \mid G \perp f, \text{ any } y \in C\} , \end{aligned} \tag{9}$$

where, e.g., $g \perp F$ is lifted pointwise[3]. We can now define that $(F, G)_c$ is *maximal* iff $\triangle F = G$ and $\triangledown G = F$.

Definition 12 (Closure sorting). *Let C be a category, and let P be a decomposable predicate on C. The* closure sorting $\mathfrak{C}(P) \to C$ *has* objects $(F, G)_c$ *where c is an object of C and F, G are sets of morphisms of C s.t. every $f \in F$ and $g \in G$ has $\mathrm{cod}(f) = c = \mathrm{dom}(g)$ and $P(g \circ f)$. Moreover, $(F, G)_c$ must be maximal. $\mathfrak{C}(P)$ has* morphisms $k : (F, G)_c \to (H, J)_d$ *those $k : c \to d$ in C satisfying*

$$f \in F \implies k \circ f \in H \text{ and } j \in J \implies j \circ k \in G \,. \tag{10}$$

It is fairly easy to establish that $\triangle$ and $\triangledown$ form a Galois connection: $\triangle F \supseteq G$ iff $F \subseteq \triangledown G$. Thus $\triangle\triangledown$ and $\triangledown\triangle$ are indeed closure operators: $\triangle\triangledown\triangle F = \triangle F$ and $\triangledown\triangle\triangledown G = \triangledown G$. (Hence the name "closure sorting".) We can use these closure operators to "fill up" a pair $(F, G)_c$ that is not maximal, taking either $(\triangledown G, \triangle\triangledown G)_c$ or $(\triangledown\triangle F, \triangle F)_c$. This realization is crucial in establishing that the fibres of a closure sorting are lattices and, in turn, that every closure sorting satisfies the premises of Proposition 2 and Theorem 1.

Lemma 1. *Let P be a decomposable predicate on C. Then $\mathfrak{C}(P)$ (1) is safe, (2) is a weak joint opfibration, and (3) lifts P-agents at every object c of C.*

By Proposition 2 and Theorem 1, every closure sorting transfers RPOs and has semantic correspondence.

Theorem 2. *Let C be a category with RPOs, and let P be a decomposable predicate on C. Then $\mathfrak{C}(P)$ has RPOs, transfers RPOs, and has semantic correspondence up to P.*

[3] The operators $\triangle, \triangledown$ are related to the BiLog [18] adjuncts to composition.

Example 5. The sorting $\mathcal{F} \to \mathcal{C}$ of Example 3 is indeed a closure sorting for the predicate "contains at most one s"; we recover the objects 0 and 1 as $0 = (\nabla\triangle\{0\}, \triangle\{0\})$ and $1 = (\nabla\{0\}, \triangle\nabla\{0\})$. By Theorem 2, $\mathcal{F}$ has RPOs and has semantic correspondence up to that predicate.

The closure sorting answers Question 2, "how do we construct a sorting given some problem domain" by virtue of Theorem 2. Technically, the answer applies only to reactive system, but again, it is straightforwardly lifted to the setting bigraphical reactive systems; we discussed this lifting in more detail in the next section.

6 Bigraphical Reactive Systems

As mentioned in the introduction, bigraphical reactive systems are not instances of ordinary reactive systems. Because categories of bigraphs do not contain relative pushouts, Milner, Leifer, and Jensen introduced supported pre-categories [13,4,1].

Intuitively, the supported pre-category of pure bigraphs adds a notion of identity of sub-terms; the totality of identities in some term is called its support, hence "supported". For the notion of support to make sense, composition of bigraphs with overlapping support is left undefined, hence "pre-category".

For each supported pre-category S we obtain a category $[S]$ by considering equal morphisms which differ only in the particular identities chosen for their support. The support quotient functor $\eta_S : S \to [S]$ takes each supported term to such a support equivalence class. The category of abstract bigraphs arise as the quotient $[S]$ of the supported pre-category S, the concrete bigraphs.

This solution, using supported pre-categories, is widely regarded as being overly ad hoc. Following a suggestion by Leifer [13], Sassone and Sobocinski [19,20] investigated using instead first two-categories and later bi-categories as foundations for reactive systems. Unfortunately, comfortable as their results may be, no one has as yet formalised bigraphs in one of these more general settings. Hence, we stick with the present formalisation in supported pre-categories, at least for the present paper.

Fortunately, lifting Jensen's previous work and the results of the preceding sections to the setting of supported pre-categories pose no real difficulties, it only requires a lot of footwork. For want of space, we omit that footwork here; the interested reader is referred to [16]. We do, however, make the following comments.

A sorting on pure abstract bigraphs induces a sorting on the corresponding concrete bigraphs, found by taking the pullback of that sorting along the support quotient.

$$\begin{array}{ccc} F^\star & \longrightarrow & S \\ \downarrow & \lrcorner & \downarrow{\scriptstyle \eta_S} \\ F & \xrightarrow[F]{} & [S] \end{array} \tag{11}$$

Here we have concrete bigraphs S, abstract bigraphs $[S]$; $\eta_S : S \to [S]$ is the support quotient. We are given a sorting $F \to [S]$, and we find the corresponding sorting $F^\star \to S$ simply by taking the pullback (in the category of supported pre-categories) of $F \to [S]$ along η_S.

It turns out that if the original sorting functor satisfies the sufficient conditions we have seen so far, then the induced sorting functor will sustain the behavioural theory. That is, in this case, the induced sorting will preserve congruence properties and enjoy semantic correspondence. Moreover, there is essentially a bijection between the sortings of a supported pre-category (S in the above diagram) and the sortings of its support quotient ($[S]$ in the above diagram). Hence, for practical work with bigraphical models, it is sufficient to consider sortings of abstract bigraphs.

Readers who find all this a bit hand-wavy are again referred to [16], where they will find formalisation. In particular, Proposition 2, Theorem 1, and Theorem 2 of the present paper are stated and proved in the bigraphical setting there as [16, Theorem 5.26, Theorem 5.29, and Theorem 5.31]. The above-mentioned bijection theorem is [16, Theorem 5.21].

Altogether, by lifting in this manner the result of the preceding sections, we fully answer the two questions posed in the introduction: we have sufficient conditions for a sorting to preserve congruence properties and enjoy semantic correspondence, and we have a construction, the closure sorting, for constructing sortings satisfying these conditions.

We proceed to demonstrate the viability of the closure sortings by recovering Milner's local bigraphs as a particular closure sorting.

7 Local Bigraphs and Closure Sortings

In the setting of bigraphical reactive systems we have binding bigraphs [4] as a natural extension of pure bigraphs [1,4]; moreover, we have local bigraphs [10,3] as a natural extension of binding bigraphs. It has been suspected [21] that local bigraphs represent the end of this evolutionary ladder. In this section, we clarify what this evolution is and demonstrate that the closure sorting for a predicate derived from the scope rule is its natural endpoint. In the process, we prove that local bigraphs can, in a sense to be made precise, be replaced by the closure sorting for this predicate.

We do not reiterate the definition of pure and local bigraphs here. Refer to one of [1,4] for the definition of pure bigraphs; to one of [18,11] for intuition and examples of pure bigraphs; to [4] for the definition of binding bigraphs; and to [10,3] for the definition of local bigraphs.

Binding bigraphs partition the ports of a signature into the binding ports and the free ports. The intuition of binding ports is that [4, p.68]:

> "all points linked to a binding port of a node u lie inside u."

Although Milner and Jensen use the word "points" and later clarify that these may be names as well as ports, binding bigraphs are surely but a means of restricting pure bigraphs to those that only peer a binding port with ports beneath it. This condition, called the scope condition, is easily seen to be decomposable.

Definition 13 (Scope predicate). *Let Σ be a binding signature. The* Scope predicate for Σ *is a predicate $\mathbf{P}_\Sigma$ on the morphisms of $\mathbf{B}(U(\Sigma))$, that is, on the pure bigraphs over $U(\Sigma)$. For a bigraph f of $\mathbf{B}(U(\Sigma))$, we define $\mathbf{P}_\Sigma(f)$ iff whenever a binding port p on a node n is in a link, then any other port p' in that link is on a node that has n as an ancestor.*

Binding ports are intended to model binders in term languages, e.g., the binder k in the input prefix $x(k).P$ of the π-calculus [4,6]. (The introduction of binding ports also enables a more expressive definition of parametric reaction rules. We shall not discuss this application here).

The progression from binding to local bigraphs is one including more and more ways to decompose bigraphs satisfying the scope predicate P_Σ given above. The closure sorting for P_Σ is an endpoint of this progression, as it contains by definition every possible such decomposition. We discuss this progression in some detail because so reveals, we believe, the essence of binding and local bigraph. The discussion will be somewhat technical, however, so feel free to skip ahead to the paragraph titled "Replaceability".

To make sure that the scope condition is preserved by composition, binding bigraphs augment objects with locations of names. Each name x in an object (m, X) is either global or located at a specific place $i \in m$. A binding bigraph is then permitted to link only appropriately located ports or inner names to located outer names or binding ports. (When we say that something is "linked to a binding port" we actually mean peered with that binding port. Because no link can contain two binding ports, it is sound to identify an edge with a binding port linked to that edge).

Ascribing only a single place to each located name does not account for all possible decompositions of a bigraph satisfying the scope condition. For instance, the following two pure bigraphs both satisfy the scope condition, as does their composition. However, we can assign no location to x that will make f a valid binding bigraph.

$$f : (2, \{x\}) \to (1, \emptyset) = /x.\mathsf{k}_{(x)}(-_0 \mid -_1)$$
$$g : (0, \emptyset) \to (2, \{x\}) = \mathsf{h}_x \mid \mathsf{h}_x$$

To remedy this deficiency, [3] introduces local bigraphs. These assign to each name x of an interface (m, X) a subset $m' \subset m$ of places, instead of just a single place $i \in m$. The global names of binding bigraphs correspond to everywhere located names.

Local bigraphs still do not capture every possible decomposition of pure bigraphs satisfying the scope condition. To wit, consider the following two bigraphs.

$$f : \ (1, \{x, y\}) \to (1, \emptyset) = \ /x, y.-_0 \tag{12}$$
$$g : \ (1, \emptyset) \to (1, \{x, y\}) = \ \mathsf{k}_{(x)}(\mathsf{h}_y) \tag{13}$$

The interfaces of local bigraphs are simply not capable of expressing that in g, the name x can be free, as long as any context can only link it to names within the scope of the control $\mathsf{k}_{(x)}$, such as the name y in this example.

The closure sorting for the scope condition includes exactly this kind of additional interfaces, and thus allows this decomposition. For the application of local bigraphs in [10], encoding of the lambda calculus as a bigraphical reactive system, this extra flexibility is not exploited: For any term or reaction rule containing binders, the bound names will not appear in the interface. However, as we will make precise below, the extra flexibility is not harmful.

Replaceability. The closure sorting for the scope predicate is also a viable substitute for local bigraphs in the following precise sense. There exists a full embedding $\iota : \mathbf{B}(\Sigma) \to \mathfrak{C}(\mathbf{P}_\Sigma)$ of local bigraphs into the closure sorting for $\mathbf{P}_\Sigma$. This embedding witnesses $\mathbf{B}(\Sigma)$ being a sub-sorting of $\mathfrak{C}(\mathbf{P}_\Sigma)$ in the sense that it makes the following diagram commute.

$$\begin{array}{ccc} \mathbf{B}(\Sigma) & \xrightarrow{\iota} & \mathfrak{C}(\mathbf{P}_\Sigma) \\ & \searrow \quad \swarrow & \\ & \mathbf{B}(U(\Sigma)) & \end{array} \tag{14}$$

Moreover, this embedding both preserves and reflects bisimilarity for any reactive system $\mathcal{R}$ on $\mathbf{B}(\Sigma)$ and its image $\mathcal{R}$ in $\mathfrak{C}(\mathbf{P}_\Sigma)$. Indeed, ι preserves and reflects transitions, and the morphisms in the image of ι has no transitions outside of that image. In this sense, the closure sorting $\mathfrak{C}(\mathbf{P}_\Sigma)$ is a reasonable substitute for local bigraphs.

Theorem 3. *Let Σ be a binding signature, let $\mathbf{P}_\Sigma$ be the Scope predicate on $\mathbf{B}(\Sigma)$, and let $\mathcal{R}$ be a reactive system on $\mathfrak{C}(\mathbf{P}_\Sigma)$. Suppose f is an agent of $\mathcal{R}$ in $\mathbf{B}(\Sigma)$. There is a transition $\iota(f) \xrightarrow{g'} h'$ in $\mathfrak{C}(\mathbf{P}_\Sigma)$ if and only if there is a transition $f \xrightarrow{g} h$ in $\mathbf{B}(\Sigma)$ and both $\iota(g) = g'$ and $\iota(h) = h'$. It follows that for agents f, g of $\mathbf{B}(\Sigma)$ we have $f \sim g$ if and only if $\iota(f) \sim \iota(g)$.*

It follows that local bigraphs sustain the behavioural theory of pure bigraphs. Milner proved as much by hand [10,3]; now, we get the same result for free.

Corollary 1. *Let Σ be a binding signature. For any supported reactive system on $\mathbf{B}(\Sigma)$, bisimilarity on the supported transitions is a congruence.*

However, we also *expand* on Milner's results, because Theorem 1 gives us that local bigraphs have semantic correspondence up to P_Σ.

Corollary 2. *Let Σ be a binding signature. Then the sorting $\mathbf{B}(\Sigma) \to \mathbf{B}(U(\Sigma))$ respects supported $\mathbf{P}_\Sigma$-reactions and -transitions.*

The proof of Theorem 3 hinges on the following characterisation of $\mathfrak{C}(\mathbf{P}_\Sigma)$, which is interesting in its own right in so far as it tells us exactly what is missing from the interfaces of local bigraphs to allow all decompositions.

Lemma 2. *Let Σ be a binding signature, let $\mathfrak{C}(\mathbf{P}_\Sigma)$ be the closure sorting for the scope predicate $\mathbf{P}_\Sigma$, and let (m, X) be an object of $\mathbf{B}(\Sigma)$. Let $\Gamma(m, X) = \mathcal{P}(\{0, \ldots, m-1\}) + \mathcal{P}(X)$ be the set comprising subsets of places and subsets of*

names of the object (m, X). *We call maps* $\rho : X \to \Gamma(m, X)$ *s.t.* $\rho(x) \in \mathcal{P}(X)$ *implies* $x \in \rho(x)$ generating maps *for* (m, X).

The fibre of $\mathfrak{C}(\mathbf{P}_\Sigma) \to \mathbf{B}(U(\Sigma))$ *over* $(0, X)$ *is isomorphic to the partial order that has only one object. The fibre of* $\mathfrak{C}(\mathbf{P}_\Sigma)$ *over* (m, X) *for* $m > 0$ *is isomorphic to the partial order over generating maps for* (m, X)*, ordered pointwise by* $\rho(x) \sqsubseteq \varrho(x)$ *if and only if either (a)* $\rho(x) \subseteq \varrho(x) \subseteq \mathcal{P}(\{0, \ldots, m-1)$*, (b)* $\varrho(x) \subseteq \rho(x) \subseteq \mathcal{P}(X)$*, or (c)* $\rho(x) \subseteq \mathcal{P}(\{0, \ldots, m-1\})$ *and* $\varrho(x) \subseteq \mathcal{P}(X)$.

The generating maps $\rho : X \to \Gamma(m, X)$ cover exactly the decomposition in (13), with the intuition that if $\rho(x) \in \mathcal{P}(X)$ then x is a name that occurs in a binder that may be safely linked to names in $\rho(x)$.

We recover the interfaces of local bigraphs as the generating maps $\rho : X \to \mathcal{P}(\{0, \ldots, m-1\})$ that take every name to a left inject, that is, that assigns locations to names.

We have now shown how local bigraphs arise as a special case closure sorting. In [16] it was shown how Milner's homomorphic sorting [1] also arise as a special case on the closure sorting.

8 Conclusion and Future Work

First, we have given a sufficient condition for a sorting to reflect reactive and transition semantics of well-sorted terms. Second, we have extended the theory of sortings for reactive systems with a new construction of sortings for decomposable predicates, the closure sorting. Third, we have sketched a generalisation of the theory of sortings for reactive systems to the setting of supported precategories. Finally, we proved that local bigraphs arise naturally as a sub-sorting of the closure sorting obtained from the scope condition. Besides alleviating the need for redeveloping the behavioural theory for local bigraphs, it supports local bigraphs as the natural extension of bigraphs with local names. We conjecture that the sortings [1,2,3,4,5,6,9,10,11] can all be obtained as closure sortings. (Of the sortings mentioned in the introduction, this list leaves out only the edge-sortings of [7], which does not appear to approximate predicates).

We see two main avenues of future work. One is to investigate sortings in other frameworks, in particular within graph rewriting [22,23] and the 2-categorical approach to reactive systems [20,19]. Another is to investigate the algebraic properties of sortings and if the closure sorting is somehow universal among sortings that capture a decomposable predicate and respect the behavioural theory.

References

1. Milner, R.: Pure bigraphs: Structure and dynamics. Information and Computation 204(1), 60–122 (2006)
2. Milner, R., Leifer, J.J.: Transition systems, link graphs and Petri nets. Technical Report 598, U. of Cambridge Computer Laboratory (2004)

3. Milner, R.: Bigraphs whose names have multiple locality. Technical Report 603, U. of Cambridge Computer Laboratory (2004)
4. Jensen, O.H., Milner, R.: Bigraphs and mobile processes (revised). Technical Report 580, U. of Cambridge Computer Laboratory (2004)
5. Milner, R.: Bigraphs for petri nets. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 686–701. Springer, Heidelberg (2004)
6. Jensen, O.H.: Mobile Processes in Bigraphs. PhD thesis, U. of Aalborg (forthcoming 2008)
7. Bundgaard, M., Sassone, V.: Typed polyadic pi-calculus in bigraphs. In: PPDP 2006, pp. 1–12 (2006)
8. Grohmann, D., Miculan, M.: Reactive systems over directed bigraphs. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 380–394. Springer, Heidelberg (2007)
9. Bundgaard, M., Hildebrandt, T.: Bigraphical semantics of higher-order mobile embedded resources with local names. In: GT-VC 2005. ENTCS, vol. 154, pp. 7–29 (2006)
10. Milner, R.: Local bigraphs and confluence: Two conjectures. In: EXPRESS 2006, pp. 42–50 (2006)
11. Birkedal, L., Debois, S., Elsborg, E., Hildebrandt, T., Niss, H.: Bigraphical Models of Context-aware Systems. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921. Springer, Heidelberg (2006)
12. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000)
13. Leifer, J.J.: Operational Congruences for Reactive Systems. PhD thesis, U. of Cambridge Computer Laboratory and Trinity College (2001)
14. Sewell, P.: From rewrite rules to bisimulation congruences. Theoretical Computer Science 274(1–2), 183–230 (2002)
15. 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)
16. Debois, S.: Sortings and Bigraphs. PhD thesis, IT University of Copenhagen (2008), http://www.itu.dk/people/debois/pubs/thesis.pdf
17. Leifer, J.J.: Synthesising labelled transitions and operational congruences in reactive systems, part 2. Technical Report RR-4395, INRIA (2002)
18. Conforti, G., Macedonio, D., Sassone, V.: Spatial logics for bigraphs. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 766–778. Springer, Heidelberg (2005)
19. Sassone, V., Sobocinski, P.: Deriving bisimulation congruences: 2-categories vs. precategories. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 409–424. Springer, Heidelberg (2003)
20. Sassone, V., Sobocinski, P.: Reactive systems over cospans. In: LICS 2005, pp. 311–320. IEEE, Los Alamitos (2005)
21. Milner, R.: Personal communication (2006)
22. Bonchi, F., Gadducci, F., König, B.: Process bisimulation via a graphical encoding. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 168–183. Springer, Heidelberg (2006)
23. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Mathematical Structures in Computer Science 16(6), 1133–1163 (2006)

Dynamic Partial Order Reduction Using Probe Sets*

Harmen Kastenberg and Arend Rensink

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands

Abstract. We present an algorithm for partial order reduction in the context of a countable universe of deterministic actions, of which finitely many are enabled at any given state. This means that the algorithm is suited for a setting in which resources, such as processes or objects, are dynamically created and destroyed, without an *a priori* bound. The algorithm relies on abstract enabling and disabling relations among actions, rather than associated sets of concurrent processes. It works by selecting so-called *probe sets* at every state, and backtracking in case the probe is later discovered to have missed some possible continuation.

We show that this improves the potential reduction with respect to persistent sets. We then instantiate the framework by assuming that states are essentially sets of *entities* (out of a countable universe) and actions test, delete and create such entities. Typical examples of systems that can be captured in this way are Petri nets and (more generally) graph transformation systems. We show that all the steps of the algorithm, including the estimation of the missed actions, can be effectively implemented for this setting.

1 Introduction

Explicit state model checking is, by now, a well-established technique for verifying concurrent systems. A strong recent trend is the extension of results to *software* systems. Software systems have, besides the problems encountered in the traditional concurrent automata, the additional problem of unpredictable dynamics, for instance in the size of the data structures, the depth of recursion and the number of threads.

Typically, the number of components in concurrent software systems is fairly large, and the actions performed by those components, individually or together (in case of synchronization), can be interleaved in many different ways. This is the main cause of the well-known *state space explosion problem* model checkers have to cope with. A popular way of tackling this problem is by using so-called *partial order reduction*. The basic idea is that, in a concurrent model of system behaviour based on *interleaving* semantics, different orderings of independent actions, e.g., steps taken by concurrent components, can be treated as *equivalent*, in which case not all possible orderings need to be explored.

In the literature, a number of algorithms have been proposed based on this technique; see, e.g. [2, 3, 4, 12, 13]. These are all based upon variations of two core techniques:

* This work has been carried out in the context of the GROOVE project funded by the Dutch NWO (project number 612.000.314).

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 233–247, 2008.

persistent (or *stubborn*) *sets* [3,12] and *sleep sets* [3]. In their original version, these techniques are based on two important assumptions:

1. The number of actions is finite and *a priori* known.
2. The system consists of a set of concurrent processes; the orderings that are pruned away all stem from interleavings of actions from distinct processes.

Due to the dynamic nature of software, the domain of (reference) variables, the identity of method frames and the number of threads are all impossible to establish beforehand; therefore, the number of (potential) actions is unbounded, meaning that assumption 1 is no longer valid. This has been observed before by others, giving rise to the development of *dynamic* partial order reduction; e.g., [2,5]. As for assumption 2, there are types of formalism that do not rely on a pre-defined set of parallel processes but which do have a clear notion of independent actions. Our own interest, for example, is to model check graph transformation systems (cf. [7,11]); here, not only is the size of the generated graphs unbounded (and so assumption 1 fails) but also there is no general way to interpret such systems as sets of concurrent processes, and so assumption 2 fails as well.

In this paper, we present a new technique for partial order reduction, called *probe sets*, which is different from persistent sets and sleep sets. Rather than on concurrent processes, we rely on abstract *enabling* and *disabling* relations among actions, which we assume to be given somehow. Like persistent sets, probe sets are subsets of enabled actions satisfying particular local (in)dependence conditions. Like the existing dynamic partial order reduction techniques, probe sets are optimistic, in that they underestimate the paths that have actually to be explored to find all relevant behaviour. The technique is therefore complemented by a procedure for identifying *missed actions*.

We show that probe set reduction preserves all traces of the full transition system system modulo the permutation of independent actions. Moreover, we show that the probe set technique is capable of reducing systems in which there are no non-trivial persistent sets, and so existing techniques are bound to fail.

However, the critical part is the missed action analysis. In principle, it is possible to miss an action whose very existence is unknown. To show that the detection of such missed actions is nevertheless feasible, we further refine our setting by assuming that actions work by manipulating (reading, creating and deleting) *entities*, in a rule-based fashion. For instance, in graph transformation, the entities are graph nodes and edges. Thus, the actions are essentially rule applications. Missed actions can then be conservatively predicted by overestimating the applicable rules.

The paper is structured as follows. In Section 2, we introduce an abstract framework for enabling and disabling relations among actions in a transition system. In Section 3 we define missed actions and probe sets, give a first version of the algorithm and establish the correctness criterion. Section 4 then discusses how to identify missed actions and construct probe sets, and gives the definitive version of the algorithm. All developments are illustrated on the basis of a running example introduced in Section 2. Section 5 contains an evaluation and discussion of related and future work.

2 Enabling, Disabling and Reduction

Throughout this paper, we assume a countable universe of *actions* Act, ranged over by $a, b, \ldots$, with two binary relations, an irreflexive relation $\triangleright$ and a reflexive relation $\blacktriangleleft$:

Stimulation: $a \triangleright b$ indicates that a stimulates b;
Disabling: $a \blacktriangleleft b$ indicates that a disables b.

The intuition is that a stimulates b if the effect of a fulfills part of the precondition of b that was not fulfilled before (meaning that b cannot occur directly before a), whereas it disables b if it violates part of b's precondition (meaning that b cannot occur directly after a). If b neither is stimulated by a nor disables a then it is independent of a (meaning that it might occur concurrently with a). In the theory of *event structures* (e.g., [14]), $\triangleright$ roughly corresponds to a notion of (direct) *cause* and $\blacktriangleleft$ to *asymmetric conflict* (e.g., [8]; see also [6] for a systematic investigation of event relations).[1] Fig. 1 shows an example.

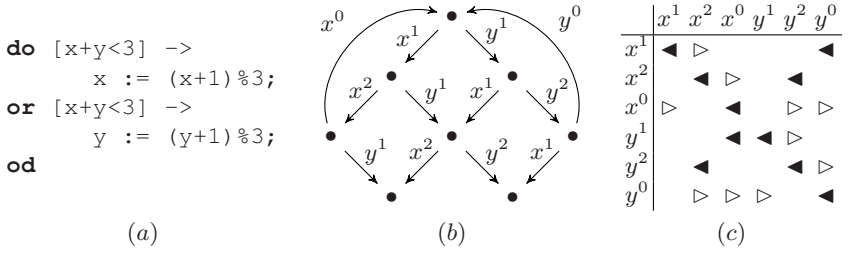


Fig. 1. A non-deterministic process (a), its transition system (b) and the stimulus and disabling relations (c). Action x^i $[y^i]$ assigns i to x $[y]$, with pre-condition $x + y < 3$.

We also use *words*, or sequences of actions, denoted $v, w \in \mathsf{Act}^*$. The empty word is denoted ε. The set of actions in w is denoted A_w. With respect to stimulation and disabling, not all words are possible computations. To make this precise, we define a derived *influence* relation over words:

$$v \rightsquigarrow w \quad :\Leftrightarrow \quad \exists a \in A_v, b \in A_w : a \triangleright b \vee a \blacktriangleright b.$$

(where $a \blacktriangleright b$ is equivalent to $b \blacktriangleleft a$.) $v \rightsquigarrow w$ is pronounced "v influences w." Influence can be positive or negative. For instance, in Fig. 1, $x^1 \cdot x^2 \rightsquigarrow y^1 \cdot y^2$ due to $x^2 \blacktriangleright y^2$ and $x^1 \cdot y^1 \rightsquigarrow x^2 \cdot y^2$ due to $x^1 \triangleright x^2$, whereas x^1 and $y^1 \cdot y^2$ do not influence one another.

Definition 1 (word feasibility). *A word w is* feasible *if*

- *for all sub-words $a \cdot v \cdot b$ of w, if $a \blacktriangleleft b$ then $\exists c \in A_v : a \triangleright c \triangleright b$;*
- *for all sub-words $v_1 \cdot v_2$ of w, $v_2 \rightsquigarrow v_1$ implies $v_1 \rightsquigarrow v_2$.*

[1] This analogy is not perfect, since in contrast to actions, events can occur only once.

The intuition is that infeasible words do not represent possible computations. For instance, if $a \blacktriangleleft b$, for the action b to occur after a, at least one action in between must have "re-enabled" b; and if v_2 occurs directly after v_1, but v_1 does *not* influence v_2, then v_2 might as well have occurred before v_1; but this also rules out that v_2 influences v_1.

For many purposes, words are interpreted up to permutation of independent actions. We define this as a binary relation over words.

Definition 2 (equality up to permutation of independent actions). $\simeq \subseteq \mathsf{Act}^* \times \mathsf{Act}^*$ *is the smallest transitive relation such that* $v{\cdot}a{\cdot}b{\cdot}w \simeq v{\cdot}b{\cdot}a{\cdot}w$ *if* $a \not\leadsto b$.

Some properties of this equivalence, such as the relation with feasibility, are expressed in the following proposition.

Proposition 3

1. *If* v *is feasible and* $v \simeq w$, *then* w *is feasible;*
2. $\simeq$ *is symmetric over the set of feasible words.*
3. $v{\cdot}w_1 \simeq v{\cdot}w_2$ *if and only if* $w_1 \simeq w_2$.

It should be noted that, over feasible words, the setup now corresponds to that of (Mazurkiewicz) *traces*, which have a long tradition; see, e.g., [1, 9]. The main difference is that our underlying notion of influence, built up as it is from stimulation and disabling, is more involved than the symmetric binary dependency relation that is commonly used in this context — hence for instance the need here to restrict to feasible words before $\simeq$ is symmetric.

We also define two prefix relations over words, the usual "hard" one ($\preceq$), which expresses that one word is equal to the first part of another, and a "weak" prefix ($\precsim$) up to permutation of independent actions. It is not difficult to see that, over feasible words, both relations are partial orders.

$$v \preceq w \;:\Leftrightarrow\; \exists u : v{\cdot}u = w \tag{1}$$

$$v \precsim w \;:\Leftrightarrow\; \exists u : v{\cdot}u \simeq w. \tag{2}$$

2.1 Transition Systems

We deal with transition systems labelled by Act. As usual, transitions are triples of source state, label and target state, denoted $q \xrightarrow{a} q'$. We use $q_0 \xrightarrow{w} q_{n+1}$ with $w = a_0 \cdots a_n$ as an abbreviation of $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} q_{n+1}$. Formally:

Definition 4. *A transition system is a tuple* $S = \langle Q, \rightarrow, \iota \rangle$ *such that* $\iota \in Q$ *and* $\rightarrow \subseteq Q \times \mathsf{Act} \times Q$, *with the additional constraints that for all* $q, q_1, q_2 \in Q$:

- *All traces are feasible; i.e.,* $\iota \xrightarrow{w} q$ *implies* w *is feasible;*
- *The system is deterministic up to independence; i.e.,* $q \xrightarrow{w_1} q_1$ *and* $q \xrightarrow{w_2} q_2$ *with* $w_1 \simeq w_2$ *implies* $q_1 = q_2$.
- *All out-degrees are finite; i.e.,* $enabled(q) = \{a \mid \exists q \xrightarrow{a} q'\}$ *is a finite set.*

The second condition implies (among other things) that the actions in Act are fine-grained enough to deduce the successor state of a transition entirely from its source state

and label. Although this is clearly a restriction, it can always be achieved by including enough information into the actions. Some more notation:

$$q \vdash w \;:\Leftrightarrow\; \exists q' : q \xrightarrow{w} q'$$
$$q \uparrow w \;:=\; q' \quad \text{such that } q \xrightarrow{w} q'.$$

$q \vdash w$ expresses that q *enables* w, and $q \uparrow w$ is q *after* w, i.e., the state reached from q after w has been performed. Clearly, $q \uparrow w$ is defined (uniquely, due to determinism) iff $q \vdash w$. In addition to determinism modulo independence, the notions of stimulation and disabling have more implications on the transitions of a transition system. These implications are identified in the following definition.

Definition 5 (dependency consistency and completeness). *A transition system S is called* dependency consistent *if it satisfies the following properties for all $q \in Q$:*

$$q \vdash a \;\wedge\; a \triangleright b \Longrightarrow q \not\vdash b \tag{3}$$
$$q \vdash a \;\wedge\; a \blacktriangleleft b \Longrightarrow q \not\vdash a{\cdot}b. \tag{4}$$

S is called dependency complete *if it satisfies:*

$$q \vdash a{\cdot}b \;\wedge\; a \not\triangleright b \Longrightarrow q \vdash b \tag{5}$$
$$q \vdash a \;\wedge\; q \vdash b \;\wedge\; a \not\blacktriangleleft b \Longrightarrow q \vdash a{\cdot}b. \tag{6}$$

Dependency consistency and completeness are illustrated in Fig. 2.

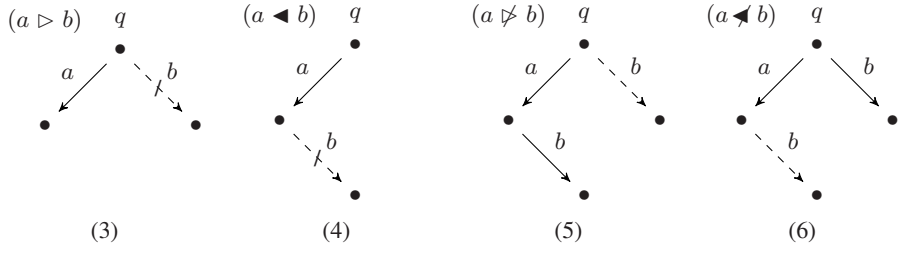


Fig. 2. The consistency and completeness properties of Def. 5. The (negated) dashed arrows are implied by the others, under the given dependency relations.

The following property states an important consequence of dependency completeness, namely that weak prefixes of traces are themselves also traces. (Note that this does not hold in general, since weak prefixes allow reshuffling of independent actions).

Proposition 6. *If S is a dependency complete transition system, then $q \vdash w$ implies $q \vdash v$ for all $q \in Q$ and $v \precsim w$.*

The aim of this paper is to *reduce* a dependency complete transition system to a smaller transition system (having fewer states and transitions), which is no longer dependency complete but from which the original transition system can be reconstructed by completing it w.r.t. (5) and (6). We now define this notion of reduction formally.

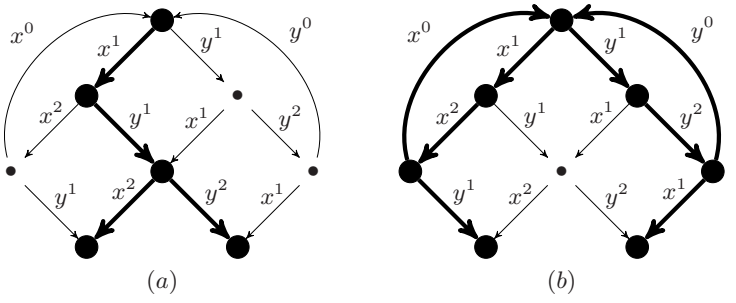


Fig. 3. An incorrect (a) and a correct (b) reduction of the transition system in Fig. 1. The fat nodes and arrows are the states and transitions of the reduced system.

Definition 7 (reduction). *Let R, S be two dependency consistent transition systems. We say that R* reduces *S if $Q_R \subseteq Q_S$, $T_R \subseteq T_S$, $\iota_R = \iota_S$, and for all $w \in \mathsf{Act}^*$*

$$\iota_S \vdash_S w \implies \exists v \in \mathsf{Act}^* : w \precsim v \ \wedge \ \iota_R \vdash_R v.$$

We will often characterise a reduced transition system only through its set of states Q_R.

For example, Fig. 3 shows two reductions of the transition system in Fig. 1, one invalid (a) and one valid (b). In (a), among others the trace $x^1 \cdot x^2 \cdot x^0$ is lost.

It follows from Proposition 6 that the reduction of a dependency complete transition system is essentially lossless: if R reduces S and S is complete, then the reachable part of S can be reconstructed from R up to isomorphism. In particular, it immediately follows that deadlock states are preserved by reduction:

Proposition 8. *If R, S are dependency consistent transition systems such that S is dependency complete and R reduces S, then for any reachable deadlock state $q \in Q_S$ (i.e., such that $\forall a \in \mathsf{Act} : q \not\vdash a$) it holds that $q \in Q_R$.*

2.2 Entity-Based System Specifications

Above we have introduced a very abstract notion of actions and dependencies. We will now show a way to instantiate this framework. In the following, Ent is a countable universe of *entities*, ranged over by $e, e_1, e', \ldots$

Definition 9. *An action a is said to be* Ent-based *if there are associated finite disjoint sets*

- $R_a \subseteq \mathsf{Ent}$, *the set of entities* read *by a;*
- $N_a \subseteq \mathsf{Ent}$, *the set of entities* forbidden *by a;*
- $D_a \subseteq \mathsf{Ent}$, *the set of entities* deleted *by a;*
- $C_a \subseteq \mathsf{Ent}$, *the set of entities* created *by a.*

The set of Ent-based actions is denoted $\mathsf{Act}[\mathsf{Ent}]$. For Ent-based actions a, b we define

$$a \rhd b :\Leftrightarrow C_a \cap (R_b \cup D_b) \neq \emptyset \vee D_a \cap (C_b \cup N_b) \neq \emptyset \tag{7}$$

$$a \lhd b :\Leftrightarrow D_a \cap (R_b \cup D_b) \neq \emptyset \vee C_a \cap (C_b \cup N_b) \neq \emptyset \tag{8}$$

Since Ent and Act may both be infinite, we have to impose some restrictions to make sure that our models are effectively computable. For this purpose we make the following important assumption:

Enabling is finite. *For every finite set $E \subseteq \mathsf{Ent}$, the set of potentialle applicable actions $\{a \in \mathsf{Act} \mid R_a \cup D_a \subseteq E\}$ is finite.*

A transition system S is called Ent-based if $A \subseteq \mathsf{Act}[\mathsf{Ent}]$ and for every $q \in Q$ there is an associated finite set $E_q \subseteq \mathsf{Ent}$, such that $E_q = E_{q'}$ implies $q = q'$.

Definition 10 (Entity-based transition systems). *A transition system S is called* Ent*-based if all transitions are labelled by* Ent*-based actions, and for all $q \in Q$:*

- *There is a finite set $E_q \subseteq \mathsf{Ent}$, such that $E_q = E_{q'}$ implies $q = q'$;*
- *For all $a \in \mathsf{Act}[\mathsf{Ent}]$, $q \vdash a$ iff $(R_a \cup D_a) \subseteq E_q$ and $(N_a \cup C_a) \cap E_q = \emptyset$;*
- *For all $a \in enabled(q)$, $q \uparrow a$ is determined by $E_{q\uparrow a} = (E_q \setminus D_a) \cup C_a$.*

It can be shown that these three conditions on the associated events, together with the assumption that enabling is computable, actually imply feasibility, determinism and finite out-degrees. The following (relatively straightforward) proposition states that this setup guarantees some further nice properties.

Proposition 11. *Every* Ent*-based transition system is dependency complete and consistent, and has only feasible words as traces.*

Models whose behaviour can be captured by entity-based transition systems include: Turing machines (the entities are symbols at positions of the tape), Petri nets (the entities are tokens), term and graph rewrite systems (the entities are suitably represented subterms and graph elements, respectively). Computability of enabling is guaranteed by the *rule-based* nature of these models: all of them proceed by attempting to instantiate a finite set of rules on the given finite set of entities, and this always results in a finite, computable set of rule applications, which constitute the actions.

For instance, the transition system of Fig. 1 is obtained (*ad hoc*) by using entities $e_{x>0}$, $e_{x>1}$, $e_{y>0}$ and $e_{y>1}$, setting $E_\iota = \emptyset$ and defining the actions as follows:

a	R_a	N_a	D_a	C_a
x^1				$e_{x>0}$
x^2	$e_{x>0}$	$e_{y>0}$		$e_{x>1}$
x^0		$e_{y>1}$	$e_{x>0}, e_{x>1}$	

a	R_a	N_a	D_a	C_a
y^1				$e_{y>0}$
y^2	$e_{y>0}$	$e_{x>0}$		$e_{y>1}$
y^0		$e_{x>1}$	$e_{y>0}, e_{y>1}$	

(9)

3 Missed Actions and Probe Sets

All *static* partial order reduction algorithms explore subsets of enabled transitions in such a way that they guarantee *a priori* not to rule out any relevant execution path of the system. *Dynamic* partial order reduction algorithms, such as e.g. [2], on the other hand, potentially "miss" certain relevant execution paths. These missed paths then have to be added at a later stage. The resulting reduction may be more effective, but additional resources (i.e. time and memory) are needed for the analysis of missed execution paths.

For instance, in the reduced system of Fig. 3(a), the transitions are chosen such that all actions that run the danger of becoming disabled are explored. Nevertheless, actions x^0 and y^0 are missed because they have never become enabled.

Our dynamic partial order reduction algorithm selects the transitions to be explored on the basis of so-called *probe sets*. We will now introduce the necessary concepts.

3.1 Missed Actions

We define a *vector* in a transition system as a tuple consisting of a state and a trace leaving that state. Vectors are used especially to characterise their *target* states, in such a way that not only the target state itself is uniquely identified (because of the determinism of the transition system) but also the causal history leading up to that state.

Definition 12 (vector). *A vector (q, w) of a transition system S consists of a state $q \in Q$ and a word w such that $q \vdash w$.*

Missed actions are actions that would have become enabled along an explored execution path if the actions in the path had been explored in a different order. To formalise this, we define the (weak) *difference* between words, which is the word that has to be concatenated to one to get the other (modulo independence), as well as the *prime cause* within w of a given action a, denoted $\downarrow_a w$, which is the smallest weak prefix of w that includes all actions that influence a, directly or indirectly:

$$w - v := u \text{ such that } v{\cdot}u \simeq w$$
$$\downarrow_a w := v \text{ such that } w - v \not\rightarrow a \ \wedge\ \forall v' \precsim w : (w - v' \not\rightarrow a \ \Rightarrow\ v \precsim v').$$

Clearly, $w - v$ exists if and only if $v \precsim w$; in fact, as a consequence of Proposition 3.3, it is then uniquely defined up to $\simeq$. The prime cause $\downarrow_a w$, on the other hand, is always defined; the definition itself ensures that it is unique up to $\simeq$. A representative of $\downarrow_a w$ can in fact easily be constructed from w by removing all actions, starting from the tail and working towards the front, that do not influence either a or any of the actions *not* removed. For instance, in Fig. 1 we have $x^1{\cdot}y^1{\cdot}y^2 - y^1 = x^1{\cdot}y^2$ whereas $x^1{\cdot}y^1{\cdot}y^2 - y^2$ is undefined; furthermore, $\downarrow_{y^2} x^1{\cdot}y^1 = y^1$.

Definition 13 (missed action). *Let (q, w) be a vector. We say that an action a is* missed along *(q, w) if $q \not\vdash w{\cdot}a$ but $q \vdash v{\cdot}a$ for some $v \precsim w$. The missed action is characterised by $\downarrow_a v{\cdot}a$ rather than just a; i.e., we include the prime cause. A missed action is said to be* fresh *in (q, w) if $w = w'{\cdot}b$ and a is not a missed action in (q, w').*

The set of fresh missed actions along (q, w) is denoted $fma(q, w)$. It is not difficult to see that $v{\cdot}a \in fma(q, w)$ implies $w = w'{\cdot}b$ such that $b \rightsquigarrow a$; otherwise a would already have been missed in (q, w').

A typical example of a missed action is $(y^1{\cdot}y^2) \in fma(\iota, x^1{\cdot}x^2{\cdot}y^1)$ in Fig. 1: here $\iota \not\vdash x^1{\cdot}x^2{\cdot}y^1{\cdot}y^2$ but $\iota \vdash y^1{\cdot}y^2$ with $y^1 \precsim x^1{\cdot}x^2{\cdot}y^1$. Note that indeed $y^1 \rhd y^2$.

3.2 Probe Sets

The most important parameter of any partial order reduction is the selection of a (proper) subset of enabled actions to be explored. For this purpose, we define so-called *probe*

Algorithm 1. Probe set based partial order reduction, first version

let $Q \leftarrow \emptyset$; *// result set of states, initialised to the empty set*
let $C \leftarrow \{(\iota_S, \varepsilon)\}$; *// set of continuations, initialised to the start state*
while $C \neq \emptyset$ **do** *// continue until there is nothing left to do*
  choose $(q, w) \in C$; *// arbitrary choice of next continuation*
  let $C \leftarrow C \setminus \{(q, w)\}$;
  if $q \uparrow w \notin S$ **then** *// test if we saw this state before*
    let $Q \leftarrow Q \cup \{q \uparrow w\}$; *// if not, add it to the result set*
    for all $v{\cdot}m \in fma(q, w)$ **do** *// identify the fresh missed actions*
      let $Q \leftarrow Q \cup \{q \uparrow v' \mid v' \preceq v\}$; *// add intermediate states*
      let $C \leftarrow C \cup \{(q \uparrow v{\cdot}m, \varepsilon)\}$; *// add a continuation*
    end for
    choose $p \in \mathcal{P}_{q,w}$; *// choose a probe set for this continuation*
    let $C \leftarrow C \cup \{(q \uparrow p(a), w{\cdot}a - p(a)) \mid a \in dom(p)\}$; *// add next continuations*
  end if
end while

sets, based on the disabling among the actions enabled at a certain state (given as the target state $q \uparrow w$ of a vector (q, w)). Furthermore, with every action in a probe set, we associate a part of the causal history that can be discharged when exploring that action. (Thus, our probe sets are actually partial functions).

Definition 14 (probe set). *For a given vector (q, w), a* probe set *is a partial function p: $enabled(q \uparrow w) \rightharpoonup \mathsf{Act}^*$ mapping actions enabled in $q \uparrow w$ onto words, such that the following conditions hold:*

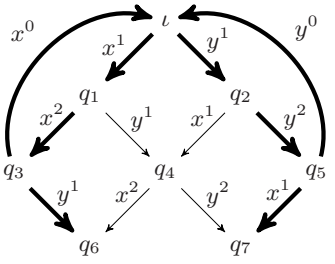
1. *For all $a \in dom(p)$ and $b \in enabled(q \uparrow w)$, $b \blacktriangleleft a$ implies $b \in dom(p)$;*
2. *For all $a \in dom(p)$ and $b \in enabled(q \uparrow w)$, $p(a) \not\preceq \downarrow_b w$ implies $b \in dom(p)$;*
3. *For all $a \in dom(p)$, $p(a) \preceq \downarrow_a w$.*

We use $\mathcal{P}_{q,w}$ to denote the set of all probe sets for a vector (q, w). We say that an action a is *in* the probe set p if $a \in dom(p)$. The first condition states that probe sets are closed under inverse disabling. The second and third conditions govern the discharge of the causal history: the fragment that can be discharged must be contained in the prime cause of any action not in the probe set (Clause 2) and of a itself (Clause 3). Of these, Clause 2 is the most involved: if we discharge any action that does not contribute to (the cause of) some b, then we must probe b as well, so that missed actions stimulated by b can still be identified. Section 4.3 gives some guidelines on how to select probe sets.

Algorithm 1 gives a first version of the reduction algorithm. In Fig. 4 we apply this algorithm to the example system of Fig. 1, obtaining the reduced system in Fig. 3(b). The first column shows the value of C at the beginning of the loop; the second column represents the choice of continuation; the third is the resulting set of fresh missed actions; the fourth column gives the increase in the result set Q; and the final column shows the choice of probe set.

3.3 Correctness

In order to have a correct reduction of a transition system, we must select sufficiently many probe sets and take care of the missing actions. Let us define this formally.



	x^1	x^2	x^0	y^1	y^2	y^0
x^1	◀	▷				◀
x^2		◀	▷		◀	
x^0	▷		◀		▷	▷
y^1			◀	◀	▷	
y^2		◀			◀	▷
y^0		▷	▷	▷		◀

iteration	C	$(q,v) \in C$	$fma(q,v)$	ΔQ	$p \in \mathcal{P}_{q,v}$
1	(ι, ε)			ι	(x^1, ε)
2	(ι, x^1)	(ι, x^1)		q_1	(x^2, ε)
3	$(\iota, x^1{\cdot}x^2)$	$(\iota, x^1{\cdot}x^2)$		q_3	$(x^0, x^1{\cdot}x^2), (y^1, \varepsilon)$
4	$(q_3, x^0), (\iota, x^1{\cdot}x^2{\cdot}y^1)$	(q_3, x^0)			
5	$(\iota, x^1{\cdot}x^2{\cdot}y^1)$	$(\iota, x^1{\cdot}x^2{\cdot}y^1)$	$y^1{\cdot}y^2$	q_6, q_2	
6	(q_5, ε)	(q_5, ε)		q_5	$(x^1, \varepsilon), (y^0, \varepsilon)$
7	$(q_5, x^1), (q_5, y^0)$	(q_5, x^1)		q_7	
8	(q_5, y^0)	(q_5, y^0)			

Fig. 4. Step-by-step execution of Algorithm 1, on the example of Fig. 1

Definition 15 (probing). *Let S be a transition system. A* probing *for S is a K-indexed set $P = \{p_{q,w}\}_{(q,w)\in K}$ where*

1. *K is a set of vectors of S such that $(\iota, \varepsilon) \in K$;*
2. *For all $(q,w) \in K$, $p_{q,w}$ is a probe set such that $(q \uparrow p_{q,w}(a), w{\cdot}a - p_{q,w}(a)) \in K$ for all $a \in dom(p_{q,w})$.*
3. *for all $(q,w) \in K$ and all $v{\cdot}a \in fma(q,w)$, there is a word $u \precsim w - v$ such that $(q \uparrow v{\cdot}a, u) \in K$ and $u \not\rightsquigarrow a$.*

We write $(q,w) \uparrow p(a)$ for the vector $(q \uparrow p_{q,w}(a), w{\cdot}a - p_{q,w}(a))$ in Clause 2. P is called fair *if for all $(q,w) \in K$ there is a function $n_{q,w}$: $enabled(q \uparrow w) \to \mathbb{N}$, assigning a natural number to all actions enabled in $q \uparrow w$, such that for all $a \in enabled(q \uparrow w)$, either $a \in dom(p_{q,w})$, or $n_{(q,w)\uparrow p(b)}(a) < n_{q,w}(a)$ for some $b \in dom(p)$.*

Clause 2 guarantees that, from a given probe set, all regular explored successors (of actions in the probe set) are indeed also probed; Clause 3 takes care of the missed actions. Fairness ensures that every enabled action will eventually be included in a probe set. In Section 4.3 we will show how to guarantee fairness.

The following is the core result of this paper, on which the correctness of the algorithm depends. It states that every fair probing gives rise to a correct reduction. The proof can be found in the appendix.

Theorem 16. *If P is a fair probing of a transition system S, then the transition system R characterized by $Q_R = \{q \uparrow w \mid (q,w) \in dom(P)\}$ reduces S.*

If we investigate Algorithm 1, in this light, it becomes clear that this is not yet correct. The total collection of vectors and probe sets produced by the algorithm give

rise to a correct probing in the sense of Def. 15 (where the u of Clause 3 is always set to ε), and also generates a probing; however, this probing is not fair. As a result, Algorithm 1, suffers from the so-called "ignoring problem" well-known from other partial order reductions.

4 The Algorithm

In this section, we put the finishing touch on the algorithm: ensuring fairness, identifying missed actions, and constructing probe sets. For this, we take the entity-based setting from Section 2.2.

4.1 Identifying Missed Actions

As we have discussed in Section 3, finding the missed actions $fma(q, v)$ by investigating all weak prefixes of v negates the benefits of the partial order reduction. In the the entity-based setting of Section 2.2, however, a more efficient way of identifying missed actions can be defined on the basis of an *over-approximation*. We define the over-approximation of the target state of a vector (q, w), denoted $q{\Uparrow}w$, as the union of all entities that have appeared along that vector, and the *weak enabling* of an action a by a set of entities E, denoted $E \Vdash a$, by only checking for the presence of read and deleted entities and not the absence of forbidden and created entities.

$$q{\Uparrow}w \;:=\; E_q \cup \bigcup\nolimits_{a \in A_w} C_a$$
$$E \Vdash a \;:\Leftrightarrow\; (R_a \cup D_a) \subseteq E$$

This gives rise to the set of *potentially missed actions*, which is a superset of the set of fresh missed actions.

Definition 17 (potentially missed actions). *Let $(q, w{\cdot}b)$ be a vector. Then, $a \in \mathsf{Act}$ is a* potentially missed action *if either $b \blacktriangleleft a$, or the following conditions hold:*

1. *a is weakly but not strongly enabled: $q{\Uparrow}w \Vdash a$ and $q{\uparrow}w \nvdash a$,*
2. *a was somewhere disabled: $\exists c \in A_w : c \blacktriangleleft a$;*
3. *a is freshly enabled: $b \rhd a$.*

We will use $pma(q, v)$ to denote the set of potentially missed actions in the vector (q, v). It is not difficult to see that $pma(q, v) \supseteq fma(q, v)$ for arbitrary vectors (q, v). However, even for a given $a \in pma(q, v)$ it is not trivial to establish whether it is really missed, since this still involves checking if there exists some $v' \precsim v$ with $q{\uparrow}v' \vdash a$, and we have little prior information about v'. In particular, it might be that v' is smaller than the prime cause $\downarrow_a v$. For instance, if $E_q = \{1\}$, $C_b = \{2\}$, $D_c = \{1, 2\}$ and $R_a = \{1, 2\}$ then $q \nvdash v{\cdot}a$ with $v = b{\cdot}c{\cdot}b$, and $\downarrow_a v = v$; nevertheless, there is a prefix $v' \precsim v$ such that $q \vdash v'{\cdot}a$, viz. $v' = b$.

In some cases, however, the latter question is much easier to answer; namely, if the prime cause $\downarrow_a v$ is the only possible candidate for such a v'. The prime cause can be computed efficiently by traversing backwards over v and removing all actions not (transitively) influencing a.

Definition 18 (reversing actions). *Two entity-based actions a, b are* reversing *if $C_a \cap D_b \neq \emptyset$ or $D_a \cap C_b \neq \emptyset$. A word w is said to be* reversing free *if no two actions $a, b \in A_w$ are reversing.*

We also use $rev_a(w) = \{b \in A_w \mid a, b \text{ are reversing}\}$ to denote the set of actions in a word w that are reversing with respect to a. Reversing freedom means that no action (partially) undoes the effect of another. For instance, in the example above b and c are reversing due to $C_b \cap D_c = \{1\}$, so v is not reversing free. The following result now states that for reversing free vectors, we can efficiently determine the fresh missed actions.

Proposition 19. *Let (q, v) is a vector with v reversing free.*

1. *For any action a, $q \vdash v' \cdot a$ with $v' \precsim v$ implies $v' = \downarrow_a v$.*
2. *$fma(q, v) = \{a \in pma(q, v) \mid q \vdash \downarrow_a v \cdot a\}$.*

4.2 Ensuring Fairness

To ensure that the probing we construct is fair, we will keep track of the "age" of the enabled actions. That is, if an action is *not* probed, its age will increase in the next round, and probe sets are required to include at least one action whose age is maximal. This is captured by a partial function $\alpha \colon \mathsf{Act} \rightharpoonup \mathbb{N}$. To manipulate these, we define

$$\begin{aligned} \alpha \oplus A &:= \{(a, \alpha(a) + 1) \mid a \in dom(\alpha)\} \cup \{(a, 0) \mid a \in A \setminus dom(\alpha)\} \\ \alpha \ominus A &:= \{(a, \alpha(a)) \mid a \notin A\} \\ \max \alpha &:= \{a \in dom(\alpha) \mid \forall b \in dom(\alpha) : \alpha(a) \geq \alpha(b)\} \\ A \text{ satisfies } \alpha &:\Leftrightarrow \alpha = \emptyset \text{ or } A \cap \max(\alpha) \neq \emptyset. \end{aligned}$$

So, $\alpha \oplus A$ initialises the age of the actions in A to zero, and increases all other ages; $\alpha \ominus A$ removes the actions in A from α; $\max \alpha$ is the set of oldest actions; and A satisfies the fairness criterion if it contains at least one oldest action, or α is empty.

4.3 Constructing Probe Sets

When constructing probe sets, there is a trade-off between the size of the probe set and the length of the vectors. On the one hand, we aim at minimising the size of the probe sets; on the other hand, we also want to minimise the size of the causal history. For example, probe sets consisting of pairs (a, ε) only (for which the second condition of Def. 14 is fulfilled vacuously, and the third trivially) are typically small, but then no causal history can be discharged. Another extreme case is when a probe set consists of pairs $(a, \downarrow_a w)$. In this case, the maximal amount of causal history is discharged that is still consistent with the third condition of Def. 14, but the probe set domain is likely to equal the set of enabled actions, resulting in no reduction at all.

The probe sets $p_{q,w}$ we construct will furthermore ensure that the vectors of the new continuation points are reversing free. Therefore, for every $p_{q,w}$ we additionally require that for all $a \in dom(p_{q,w}) : rev_a(w) \subseteq A_{p(a)}$. Since $rev_a(w) \subseteq A_{\downarrow_a w}$, this does not conflict with Def. 14.

Algorithm 2. Probe set based partial order reduction algorithm, definitive version.

1: **let** $Q \leftarrow \emptyset$;
2: **let** $C \leftarrow \{(\iota_S, \varepsilon, \emptyset)\}$; *// age function initially empty*
3: **while** $C \neq \emptyset$ **do**
4: **choose** $(q, w, \alpha) \in C$;
5: **let** $C \leftarrow C \setminus \{(q, w, \alpha))\}$;
6: **if** $q \uparrow w \notin S$ **then**
7: **let** $Q \leftarrow Q \cup \{q \uparrow w\}$;
8: **for all** $v{\cdot}m \in fma(q, w)$ **do** *// calculated according to Proposition 19.2*
9: **let** $Q \leftarrow Q \cup \{q \uparrow v' \mid v' \preceq v\}$;
10: **let** $C \leftarrow C \cup \{(q \uparrow v{\cdot}m, \varepsilon, \emptyset)\}$;
11: **end for**
12: **choose** $p \in \mathcal{P}_{q,w}$ such that $dom(p)$ satisfies α, and $\forall a \in dom(p) : rev_a(w) \subseteq A_{p(a)}$;
// choose a fair probe set, and ensure reversing freedom
13: **let** $\alpha \leftarrow \alpha \oplus enabled(q \uparrow w) \ominus dom(p)$; *// update the age function*
14: **let** $C \leftarrow C \cup \{(q \uparrow p(a), w{\cdot}a - p(a), \alpha) \mid a \in dom(p)\}$;
15: **end if**
16: **end while**

An interesting probe set $p_{q,w}$ could be constructed such that $p_{q,w}$ satisfies the condition on disabling actions and furthermore $p_{q,w}(a) = \downarrow_a w$ except for one action, say a', which is mapped to the empty vector, i.e. $p_{q,w}(a') = \varepsilon$. This action a' then ensures that no further action needs to be included in the probe set. The selection of this action a' can be based on the length of its prime cause within w.

There is a wide range of similar heuristics that use different criteria for selecting the first action from which to construct the probe set or for extending the causal history to be discharged. Depending on the nature of the transition system to be reduced, specific heuristics might result in more reduction. This is a matter of future experimentation.

Algorithm 2 now shows the definitive version of the algorithm. The differences with the original version are commented. Correctness is proved using Theorem 16. The proof relies on the fact that the algorithm produces a fair probing, in the sense of Def. 15.

Theorem 20. *For a transition system S, Algorithm 2 produces a set of states $Q \subseteq Q_S$ characterising a reduction of S.*

For our running example of Figs. 1 and 4, there are several observations to be made.

- The probe sets we constructed in Fig. 4 (on an ad hoc basis) are reversing free. Note that (in terms of (9)) x^0 reverses x^1 and x^2; likewise, y^0 reverses y^1 and y^2.
- The run in Fig. 4 is *not* fair: after the first step, the age of y^1 becomes 1 and hence y^1 should be chosen rather than x^2. This suggests that our method of enforcing fairness is too rigid, since the ignoring problem does not actually occur here.

5 Conclusion

Summary. We have proposed a new algorithm for dynamic partial order reduction with the following features:

- It can reduce systems that have no non-trivial persistent sets (and so traditional methods do not have an effect).
- It is based on abstract enabling and disabling relations, rather than on concurrent processes. This makes it suitable for, e.g., graph transformation systems.
- It uses a universe of actions that does not need to be finite or completely known from the beginning; rather, by adopting an entity-based model, enabled and missed actions can be computed on the fly. This makes it suitable for dynamic systems, such as software.
- It can deal with cyclic state spaces.

We have proved the algorithm correct (in a rather strong sense) and shown it on a small running example. However, an implementation is as yet missing.

Related Work. Traditional partial order reduction (see e.g. [3,12]) is based on statically determined dependency relations, e.g. for constructing *persistent sets*. More recently, dynamic partial order reduction techniques have been developed that compute dependency relations on-the-fly. In [2], for example, partial order reduction is achieved by computing persistent sets dynamically. This technique performs a stateless search, which is the key problem of applying it to cyclic state spaces. In [5], Gueta et al. introduce a *Cartesian* partial order reduction algorithm which is based on reducing the number of context switches and is shown also to work in the presence of cycles. Both approaches are based on processes or threads performing read and/or write operations on local and/or shared variables. The setting we propose is more general in the sense that actions are able to create or delete entities that can be used as communication objects. Therefore, our algorithm is better suited for systems in which resources are dynamically created or destroyed without an a priori bound.

Future Work. As yet, there is no implementation of probe sets. Now that the theoretical correctness of the approach is settled, the first step is to implement it and perform experiments. We plan to integrate the algorithm in the Groove tool set [10], which will then enable partial order reduction in the context of graph transformations. The actual reduction results need to be compared with other algorithms, by performing some benchmarks; see, e.g., [5].

In the course of experimentation, there are several parameters by which to tune the method. One of them is obviously the choice of probe sets; a discussion of the possible variation points was already given in Section 4. However, the main issue, which will eventually determine the success of the method, is the cost of backtracking necessary for repairing missed actions, in combination with the precision of our (over-)estimation of those missed actions. If the over-estimation is much too large, then the effect of the partial order reduction may be effectively negated.

To improve this precision, analogous to the over-approximation of an exploration path, an under-approximation can be used for decreasing the number of potentially missed actions. Actions that create or forbid entities that are in the under-approximation can never be missed actions and do not have to be considered. Essentially, also including this under-approximation means we are introducing a three-valued logic for determining the presence of entities.

Other issues to be investigated are the effect of heuristics such as discussed in Section 4.3, alternative ways to ensure fairness, and also the combination of our algorithm with the *sleep set* technique [3].

Acknowledgment. We want to thank Wouter Kuijper for contributing to this work in its early stages, through many discussions and by providing a useful motivating example.

References

1. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific Publishing Co., Inc., Singapore (1995)
2. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proc. of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), pp. 110–121. ACM Press, New York (2005)
3. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
4. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. Formal Methods in System Design 2(2), 149–164 (1993)
5. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bosnacki, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 5495, pp. 95–112. Springer, Heidelberg (2007)
6. Janicki, R., Koutny, M.: Structure of concurrency. Theor. Comput. Sci. 112(1), 5–52 (1993)
7. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)
8. Langerak, R.: Transformations and Semantics for LOTOS. PhD thesis, University of Twente (1992)
9. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1987)
10. Rensink, A.: The GROOVE Simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
11. Rensink, A.: Explicit state model checking for graph grammars. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Festschrift for Ugo Montanari. LNCS, vol. 5065. Springer, Heidelberg (2008)
12. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
13. Valmari, A.: On-the-fly verification with stubborn sets. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 397–408. Springer, Heidelberg (1993)
14. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)

A Space-Efficient Probabilistic Simulation Algorithm*

Lijun Zhang

Universität des Saarlandes, Saarbrücken, Germany
zhang@cs.uni-sb.de

Abstract. In the context of probabilistic automata, time efficient algorithms for probabilistic simulations have been proposed lately. The space complexity thereof is quadratic in the size of the transition relation, thus space requirements often become the practical bottleneck. In this paper, we exploit ideas from [3] to arrive at a space-efficient algorithm for computing probabilistic simulations based on partition refinement. Experimental evidence is given that not only the space-efficiency is improved drastically. The experiments often require orders of magnitude less time.

1 Introduction

Probabilistic automata (PAs) are a central model for concurrent systems exhibiting random phenomena. Not uncommon for concurrent system models, their verification often faces state space explosion problems. Probabilistic simulation [12] has been introduced to compare the stepwise behaviour of states in probabilistic automata. As in the non-probabilistic setting [9], the simulation preorder is especially important in compositional verification and model checking on probabilistic current systems.

In the non-probabilistic setting, a decision algorithm for the simulation preorder has been proposed in [7] with complexity $\mathcal{O}(mn)$ where n denotes the number of states and m denotes the number of transitions of labelled graphs. The space complexity is $\mathcal{O}(n^2)$ due to the need of saving the simulation relations. Since space could become the bottleneck in many applications [4], a space efficient algorithm has been introduced by Bustan and Grumberg [3]. With $n_\diamond$ denoting the number of simulation equivalence classes, the resulting space complexity is $\mathcal{O}(n_\diamond^2 + n \log n_\diamond)$, which can be considered optimal: the first part is needed to save the simulation preorder over the simulation equivalence classes, and the second part is needed to save to which simulation equivalence class a state belongs. The corresponding time complexity obtained is rather excessive: $\mathcal{O}(n^2 n_\diamond^2 (n_\diamond^2 + m))$. Tan and Cleaveland [13] combined the techniques in [7] with the bisimulation minimisation algorithm [10], and achieved a better time

* This work is supported by the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS and by the European Commission under the IST framework 7 project QUASIMODO.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 248–263, 2008.

complexity $\mathcal{O}(m \log n + mn_\sim)$, where $n_\sim$ denotes the number of bisimulation equivalence classes. The corresponding space complexity $\mathcal{O}(m + n_\sim^2)$.

Gentilini et al. [6] incorporated the efficient algorithm of [7] into the partition refinement scheme and achieved a better time complexity $\mathcal{O}(mn_\diamond^2)$ while keeping the optimal space complexity $\mathcal{O}(n_\diamond^2 + n \log n_\diamond)$. This is achieved by characterising a simulation relation by a partition pair, which consists of a partition of the set of states and a relation over the partition. Then, the simulation problem can be reduced to a generalised coarsest partition problem (GCPP), which consists of determining the coarsest stable partition pair. The algorithm starts with the coarsest partition pair and refines both the partition and the relation over the partition according to stability conditions. In [11], an algorithm has been proposed with time complexity $\mathcal{O}(mn_\diamond)$ and space complexity $\mathcal{O}(nn_\diamond)$. Recently, van Glabbeek and Ploeger [14] have shown that the proofs in [6] were flawed, but have provided a fix for the main result.

In the probabilistic setting, Baier *et al.* [1] introduced a polynomial decision algorithm for simulation preorder with time complexity $\mathcal{O}((mn^6 + m^2n^3)/\log n)$ and space complexity $\mathcal{O}(m^2)$, by tailoring a network flow algorithm to the problem, embedded in an iterative refinement loop. Drastic improvements are possible by observing that the networks on which the maximum flows are calculated, are very similar across iterations of the refinement loop [17,16]. By adaptation of the parametric maximum flow algorithm [5] to solve the maximum flows for the arising sequences of similar networks, an algorithm with overall time complexity $\mathcal{O}(m^2n)$ and space complexity $\mathcal{O}(m^2)$ has been introduced.

In this paper, we first discuss the smallest quotient automata induced by simulation preorder for PAs. Then, we discuss how to incorporate the partition refinement scheme into the algorithm for deciding simulation preorder. As in the non-probabilistic setting, we show first that simulation relations can also be characterised by partition pairs, thus the problem can be reduced to GCPP. Since in PAs, states have in general non-trivial distributions instead of single state as successors, a new proof technique is needed for the partition refinement scheme: In the non-probabilistic setting, edges have no labels and predecessor-based method can be used to refine the partition. This can not be extended to the probabilistic setting in an obvious way, since in PAs, states have successor distributions equipped with action labels. We propose a graph based analysis to refine the partition for PAs. As in [6], the relation over the partition is refined according to stability conditions. We arrive at an algorithm with space complexity $\mathcal{O}(n_\diamond^2 + n \log n_\diamond)$. Since PAs subsume labelled graphs, this can be considered as optimal. We get, however, a rather excessive time complexity of $\mathcal{O}(mn_\diamond + m_\diamond^2 n_\diamond^4 + m_\sim^2 n_\diamond^2)$ where $m_\sim$ denotes the number of transitions in the bisimulation quotient. Similar to algorithms for deciding simulation preorder for PAs [16], one can use parametric maximum flow techniques to improve the time complexity. However, more memory is then needed due to the storage of the networks and the maximum flow values of the corresponding networks across iterations. We show combined with parametric maximum flow techniques, our algorithm uses time $\mathcal{O}(mn_\diamond + m_\sim^2 n_\diamond^2)$ and space $\mathcal{O}(m_\diamond^2 + n \log n_\diamond)$.

We have implemented both the space-efficient and time-efficient variants of the partition refinement based algorithm. Experimental results show that the space-efficient algorithm is very effective in time and memory. Comparing to the original algorithm, not only the space-efficiency is improved drastically, often orders of magnitude less time are required. As in [2], both regular and random experiments show that the parametric maximum flow based implementation does not perform better in general.

This paper is organised as follows. After recalling some definitions in Section 2, we show in Section 3 that every probabilistic automaton has a quotient automaton which is the smallest in size, and this quotient automaton can be obtained by the simulation preorder. In Section 4, we show that simulation relations can also be characterised by partition pairs. Using this, we develop a partition refinement based algorithm for computing the simulation preorder in Section 5. Finally, we report experimental results in Section 6 and conclude the paper in Section 7. All proofs and more examples can be found in [15].

2 Preliminaries

Let AP be a fixed, finite set of atomic propositions. Let X, Y be finite sets. For $f : X \to \mathbb{R}$, let $f(A)$ denote $\sum_{x\in A} f(x)$ for all $A \subseteq X$. If $f : X \times Y \to \mathbb{R}$ is a two-dimensional function, let $f(x, A)$ denote $\sum_{y\in A} f(x, y)$ for all $x \in X$ and $A \subseteq Y$, and $f(A, y)$ denote $\sum_{x\in A} f(x, y)$ for all $y \in Y$ and $A \subseteq X$. For a finite set S, a distribution μ on S is a function $\mu : S \to [0, 1]$ satisfying the condition $\mu(S) \leq 1$. The support of μ is defined by $Supp(\mu) = \{s \mid \mu(s) > 0\}$, and the size of μ is defined by $|\mu| = |Supp(\mu)|$. Let $Dist(S)$ denote the set of distributions over the set S. We recall the definition of probabilistic automata [12]:

Definition 1. *A probabilistic automaton (PA) is a tuple $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$ where S is a finite set of states, $s_0 \in S$ is the initial state, Act is a finite set of actions, $\mathbf{P} \subseteq S \times Act \times Dist(S)$ is a finite set, called the probabilistic transition matrix, and $L : S \to 2^{AP}$ is a labelling function.*

For $(s, \alpha, \mu) \in \mathbf{P}$, we use $s \xrightarrow{\alpha} \mu$ as a shorthand notation, and call μ an α-successor distribution of s. Let $Act(s) = \{\alpha \mid \exists \mu : s \xrightarrow{\alpha} \mu\}$ denote the set of actions enabled at state s. For a set of states $B \subseteq S$, let $Act(B) = \cup_{s\in B} Act(s)$. For $s \in S$ and $\alpha \in Act(s)$, let $Steps_\alpha(s) = \{\mu \in Dist(S) \mid s \xrightarrow{\alpha} \mu\}$ and $Steps(s) = \cup_{\alpha\in Act(s)} Steps_\alpha(s)$. A state s is reachable from s_0, if there exists a sequence $(s_0, \alpha_0, \mu_0), \ldots, (s_{k-1}, \alpha_{k-1}, \mu_{k-1}), s_k$ with $s_k = s$, and $s_i \xrightarrow{\alpha_i} \mu_i$ and $\mu_i(s_{i+1}) > 0$ for $i = 0, \ldots, k - 1$.

Simulation Relations. Simulation requires that every α-successor distribution of one state has a corresponding α-successor distribution of the other state. The correspondence of distributions is naturally defined with the concept of *weight functions* [8].

Definition 2. *Let $\mu \in Dist(S), \mu' \in Dist(S')$ and $R \subseteq S \times S'$. A weight function for (μ, μ') with respect to R is a function $\Delta : S \times S' \to [0, 1]$ such that (i)*

$\Delta(s, s') > 0$ *implies* $(s, s') \in R$, *(ii)* $\mu(s) = \Delta(s, S')$ *for* $s \in S$ *and (iii)* $\mu'(s') = \Delta(S, s')$ *for* $s' \in S'$.

For $(s, s') \in R$ we write also $s\ R\ s'$. We write $\mu \sqsubseteq_R \mu'$ if there exists a weight function for (μ, μ') with respect to R. Obviously, for all $R \subseteq R'$, $\mu \sqsubseteq_R \mu'$ implies that $\mu \sqsubseteq_{R'} \mu'$. We recall first the definition of simulation relation [12] inside one PA:

Definition 3. *Let* $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$ *be a PA. The relation* $R \subseteq S \times S$ *is a* simulation *over* $\mathcal{M}$ *iff for all* s_1, s_2 *with* $s_1\ R\ s_2$ *it holds that:* $L(s_1) = L(s_2)$ *and if* $s_1 \xrightarrow{\alpha} \mu_1$ *then there exists a transition* $s_2 \xrightarrow{\alpha} \mu_2$ *with* $\mu_1 \sqsubseteq_R \mu_2$. *We write* $s_1 \precsim s_2$ *iff there exists a simulation* R *over* $\mathcal{M}$ *such that* $s_1\ R\ s_2$.

We say also that s_2 simulates s_1 in $\mathcal{M}$ iff $s_1 \precsim s_2$. The preorder $\precsim$ is the coarsest simulation relation over $\mathcal{M}$. If $s \precsim s'$ and $s' \precsim s$, we say that they are simulation equivalent, and write $s \simeq s'$. The notion of simulation relations can be lifted to the automata level.

Definition 4. *Let* $\mathcal{M}_1 = (S_1, s_1, Act_1, \mathbf{P}_1, L_1)$ *and* $\mathcal{M}_2 = (S_2, s_2, Act_2, \mathbf{P}_2, L_2)$ *be two PAs with disjoint set of states. We say that* $R \subseteq S_1 \times S_2$ *is a simulation over* $\mathcal{M}_1 \times \mathcal{M}_2$ *iff* $(s_1, s_2) \in R$ *and for all* s, s' *with* $s\ R\ s'$ *it holds that:* $L(s) = L(s')$ *and if* $s \xrightarrow{\alpha} \mu$ *then there exists a transition* $s' \xrightarrow{\alpha} \mu'$ *with* $\mu \sqsubseteq_R \mu'$. *We write* $\mathcal{M}_1 \precsim \mathcal{M}_2$ *iff there exists a simulation* R *over* $\mathcal{M}_1 \times \mathcal{M}_2$ *such that* $s_1\ R\ s_2$.

If $\mathcal{M}_1 \precsim \mathcal{M}_2$ and $\mathcal{M}_2 \precsim \mathcal{M}_1$, we say that they are simulation equivalent, and write $\mathcal{M}_1 \simeq \mathcal{M}_2$.

Partitions. A partition of S is a set Σ which consists of pairwise disjoint subsets of S such that $S = \cup_{B \in \Sigma} B$. The elements of a partition are also referred to as *blocks*. A partition Σ is *finer* than Σ' if for each block $Q \in \Sigma$ there exists a unique block $Q' \in \Sigma'$ such that $Q \subseteq Q'$. If Σ is finer than Σ', the *parent* block of $B \in \Sigma$ with respect to Σ', denoted by $Par_{\Sigma'}(B)$, is defined as the unique block $B' \in \Sigma'$ with $B \subseteq B'$. For $s \in S$, let $[s]_\Sigma$ denote the unique block in Σ containing state s. If Σ is clear from the context, we write simply $[s]$. For a distribution $\mu \in Dist(S)$ and a partition Σ over S, we define the induced lifted distribution with respect to Σ, denoted as $lift_\Sigma(\mu) \in Dist(\Sigma)$, by: $lift_\Sigma(\mu)(B) = \sum_{s \in B} \mu(s)$.

For a given PA $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$, a partition Σ over S is called consistent with respect to the labelling function L, if for all $B \in \Sigma$ and for all $s, s' \in B$ it holds that $L(s) = L(s')$. Intuitively, if Σ is consistent with respect to L, states in the same block have same labels. Recall $s \precsim s'$ implies that $L(s) = L(s')$. In this paper we consider only partitions which are consistent with respect to L. For consistent partition Σ and $B \in \Sigma$, we write $L(B)$ to denote the label of B.

The partition Σ over S induces an equivalence relation $\equiv_\Sigma$ defined by: $s \equiv_\Sigma s'$ iff $[s] = [s']$. If R is an equivalence relation, we let $S/_R$ denote the set of equivalence classes, which can also be considered a partition of S. Let $I_S = \{(s, s) \mid s \in S\}$ denotes the identity relation. For an arbitrary relation R with $I_S \subseteq R$, let R^* denote the transitive closure of it, which is a preorder. It induces an equivalence relation $\equiv_{R^*}$ defined by: $s \equiv_{R^*} s'$ if sR^*s' and $s'R^*s$. As a

shorthand notation, we let $S/_{R^*}$ denote the corresponding set of equivalence classes $S/_{\equiv_{R^*}}$.

The Quotient Automata. Let $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$ be a PA, and consider the partition Σ over S. For notational convenience, we use $\mu \in Dist(S)$ to denote a distribution over S, and $\pi_\Sigma \in Dist(\Sigma)$ to denote a lifted distribution over the partition Σ. If the partition Σ is clear from the context, we use π instead of π_Σ. For a set $B \subseteq S$, we write

- $B \xrightarrow{\alpha} \pi_\Sigma$ if there exists $s \in B$ and $s \xrightarrow{\alpha} \mu$ with $\pi_\Sigma = lift_\Sigma(\mu)$,
- $B \overset{\alpha}{\rightarrowtail} \pi_\Sigma$ if for all $s \in B$ there exists $s \xrightarrow{\alpha} \mu$ with $\pi_\Sigma = lift_\Sigma(\mu)$.

The $\exists$-quotient automaton $\exists\mathcal{M}/_\Sigma$ is the tuple $(\Sigma, B_0, Act, \mathbf{P}_\exists, L')$ where B_0 is the unique block containing the initial state s_0, and the transition matrix is defined by: $\mathbf{P}_\exists = \{(B, \alpha, \pi_\Sigma) \mid B \in \Sigma \wedge B \xrightarrow{\alpha} \pi_\Sigma\}$, and the labelling function is defined by $L'(B) = L(B)$. Note that $L'(B)$ is well defined because we have assumed that the partition Σ is consistent with respect to L. If no confusion arises, we use B both as a state in the $\exists$-quotient automaton, and as a set of states in $\mathcal{M}$.

We introduce some notations for the $\exists$-quotient automaton. For $s \in \Sigma$ and $\alpha \in Act(s)$, let $Steps_{\Sigma,\alpha}(s) = \{\pi \in Dist(\Sigma) \mid s \xrightarrow{\alpha} \mu \wedge \pi = lift_\Sigma(\mu)\}$, and for $B \in \Sigma$ let $Steps_{\Sigma,\alpha}(B) = \cup_{s \in B} Steps_{\Sigma,\alpha}(s)$.

The $\forall$-quotient automaton $\forall\mathcal{M}/_\Sigma$ is defined similarly: it is the tuple $(\Sigma, B_0, Act, \mathbf{P}_\forall, L')$ where the transition matrix is defined by: $\mathbf{P}_\forall = \{(B, \alpha, \pi_\Sigma) \mid B \in \Sigma \wedge B \overset{\alpha}{\rightarrowtail} \pi_\Sigma\}$, and B_0, L' as defined for the $\exists$-quotient automaton.

3 The Minimal Quotient Automaton

For a given PA $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$, in this section we show that there exists a PA $\mathcal{M}'$ which is simulation equivalent with $\mathcal{M}$, and $\mathcal{M}'$ is the smallest in size.

In the non-probabilistic setting [3], the notion of *little brothers* is introduced which states that state s_1 is a little brother of s_2 if they have a common predecessor s_3, and s_2 simulates s_1 but not the other way around. We lift this notion to PAs:

Definition 5. *Let $s \in S$ be a state, and let $\alpha \in Act(s)$ be an enabled action out of s. For two distributions $\mu, \mu' \in Steps_\alpha(s)$, we say that μ is a little brother of μ' if it holds that $\mu \sqsubseteq_{\precsim} \mu'$ and $\mu' \not\sqsubseteq_{\precsim} \mu$.*

By eliminating the little brothers from each state $s \in S$ in a PA we get a simulation equivalent PA:

Lemma 1. *Let $\mathcal{M}$ be a PA, and let $\mathcal{M}'$ be the PA obtained from $\mathcal{M}$ by eliminating little brothers. Then, $\mathcal{M} \simeq \mathcal{M}'$.*

Recall that the preorder $\precsim$ on S induces an equivalence relation $\simeq$. The following lemma states that $\mathcal{M}$ and its $\forall$-quotient automaton with respect to $\simeq$ are simulation equivalent.

Lemma 2. *Given a PA $\mathcal{M}$, the equivalence relation $\simeq$ over $\mathcal{M}$ induces a partition of S defined by: $\Sigma = \{\{s' \mid s' \in S \wedge s \simeq s'\} \mid s \in S\}$. Then, $\forall\mathcal{M}/_\Sigma$ and $\mathcal{M}$ are simulation equivalent: $\forall\mathcal{M}/_\Sigma \simeq \mathcal{M}$.*

Note that the $\forall$-quotient automaton can be obtained from the $\exists$-quotient automata by eliminating little brothers in it. Combining Lemma 1 and the above lemma, we have that $\mathcal{M}$, its $\forall$-quotient automaton, and its $\exists$-quotient automaton are pairwise simulation equivalent. For the PA $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$, we let $n = |S|$ denote the number of the states, and $m = \sum_{s\in S}\sum_{\alpha\in Act(s)}\sum_{\mu\in Steps_\alpha(s)}|\mu|$ denote the size of the transitions. The following lemma states that the $\forall$-quotient automaton of $\mathcal{M}$ is the smallest one among those PAs which are simulation equivalent to $\mathcal{M}$.

Lemma 3. *Let $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$ be a PA in which all states are reachable from s_0. Let $\mathcal{M}' = (S', s_0', Act, \mathbf{P}', L')$ be any other PA which is simulation equivalent with $\mathcal{M}$. Let Σ denote the partition of S induced by $\simeq$. Moreover, let m_Σ, n_Σ be the size of transitions and states of $\forall\mathcal{M}/_\Sigma$, m', n' be the size of transitions and states of $\mathcal{M}'$ respectively. Then, it holds that $n_\Sigma \leq n'$ and $m_\Sigma \leq m'$.*

In the above lemma, we require that all states in the PA are reachable from the initial state. Note this is not a real restriction. As in the non-probabilistic setting [3], by pruning the unreachable states of $\mathcal{M}$ we get a PA which is simulation equivalent to $\mathcal{M}$. Thus, to construct the minimal quotient automaton for $\mathcal{M}$, we can eliminate the unreachable states, compute the simulation preorder, and then delete the little brothers. The dominating part is to decide the simulation preorder.

4 Simulation Characterised by Partition Pairs

In the non-probabilistic setting, the simulation preorder for unlabelled graph is characterised by partition pairs [3,6] which consist of a partition of the state space and a binary relation over the partition. Then, a partition refinement approach is introduced based on partition pairs. In this section, we adapt the notion of partition pairs to PAs, and then we show that we can characterise simulation relations for PAs by partition pairs. This is the basis for the partition refinement approach which will be introduced in the next section. In the remainder of this section, we fix a PA $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$.

We say that the pair $(B, B') \in \Sigma \times \Sigma$ respects the labelling function L if $L(B) = L(B')$. Now we give the definition of partition pairs.

Definition 6 (Partition Pair). *A partition pair over S is a pair $\langle\Sigma, \Gamma\rangle$ where Σ is a partition of S, and $\Gamma \subseteq \Sigma \times \Sigma$ is a reflexive relation over Σ satisfying the condition: all pair $(B, B') \in \Gamma$ respects the labelling function L.*

We also call Γ the partition relation. Let Υ denote the set of all partition pairs over S. For $\langle\Sigma, \Gamma\rangle \in \Upsilon$ and $B, B' \in \Sigma$, we also write also $B\Gamma B'$ if $(B, B') \in \Gamma$. A partition pair induces a binary relation on S as follows:

Definition 7 (Induced Relation). *The partition pair $\langle\Sigma, \Gamma\rangle \in \Upsilon$ induces the binary relation on S by: $\precsim_{\langle\Sigma,\Gamma\rangle} = \{(s, s') \mid [s]\Gamma[s']\}$.*

Let $\langle \Sigma, \Gamma \rangle, \langle \Sigma', \Gamma' \rangle \in \Upsilon$. If Σ if finer than Σ', and $\precsim_{\langle \Sigma, \Gamma \rangle} \subseteq \precsim_{\langle \Sigma', \Gamma' \rangle}$ holds, we say that Γ is finer than Γ'. Now we introduce a partial order on Υ:

Definition 8 (Partial Order). *We define an order $\ltimes \subseteq \Upsilon \times \Upsilon$ as follows: $\langle \Sigma, \Gamma \rangle \ltimes \langle \Sigma', \Gamma' \rangle$ if Σ is finer than Σ' and Γ is finer than Γ'.*

If $\langle \Sigma, \Gamma \rangle \ltimes \langle \Sigma', \Gamma' \rangle$ we say $\langle \Sigma, \Gamma \rangle$ is finer than $\langle \Sigma', \Gamma' \rangle$. Obviously the defined relation is a partial order: $\ltimes$ satisfies the reflexivity, antisymmetry and transitivity conditions. Now we introduce the stability of partition pairs.

Definition 9 (Stable Partition Pairs). *A partition pair $\langle \Sigma, \Gamma \rangle \in \Upsilon$ is* stable *if for each $B \Gamma B'$ and $B \xrightarrow{\alpha} \pi_\Sigma$, there exists $B' \overset{\alpha}{\rightarrowtail} \pi'_\Sigma$ such that $\pi_\Sigma \sqsubseteq_\Gamma \pi'_\Sigma$.*

Let Υ_{sta} denote the set of all stable partition pairs, and let $\Upsilon^{\diamond}_{sta} \subseteq \Upsilon_{sta}$ be the set of stable partition pairs in which the partition relation is a preorder. We show that a stable partition pair induces a simulation relation.

Theorem 1 (Induced Simulation Relation). *Let $\langle \Sigma, \Gamma \rangle \in \Upsilon_{sta}$ be a stable partition pair. Then, the induced relation $\precsim_{\langle \Sigma, \Gamma \rangle}$ is a simulation relation.*

In the following we give the definition that a set of states is stable with respect to a partition pair:

Definition 10. *Let $\langle \Sigma, \Gamma \rangle$ be a partition pair and let $B \in \Sigma$. Assume that $Q \subseteq B$. We say that Q is stable with respect to $\langle \Sigma, \Gamma \rangle$ if $Q \xrightarrow{\alpha} \pi_\Sigma$ implies that there exists $Q \overset{\alpha}{\rightarrowtail} \pi'_\Sigma$ such that $\pi_\Sigma \sqsubseteq_\Gamma \pi'_\Sigma$.*

Assume that Σ' is a refinement of Σ. Then, we say that Σ' is stable with respect to $\langle \Sigma, \Gamma \rangle$ if each $B \in \Sigma'$ is stable with respect to $\langle \Sigma, \Gamma \rangle$.

Simulations & Stable Partition Pairs. We define a function which establishes connections between simulation relations and stable partition pairs. Recall that $I_S = \{(s, s) \mid s \in S\}$ denotes the identity relation over S. We consider the set $\Xi := \{R \subseteq S \times S \mid I_S \subseteq R\}$ of relations containing the identity relation. We define the function $H : \Xi \to \Upsilon$ by: $H(R) = (S/_{R^*}, \Gamma_R)$ where Γ_R is defined by: $B \Gamma_R B'$ if $s R^* s'$ for all $s \in B$ and $s' \in B'$. For $R \in \Xi$, $H(R)$ is a partition pair where the partition is induced by $\equiv_{R^*}$, and $B \Gamma_R B'$ if states in B' are reachable from states in B in the transitive closure R^*. If R is a simulation relation, $s R^* s'$ implies that $L(s) = L(s')$ which implies that Γ_R respects the labelling function. The following lemma states that for a preorder and a simulation relation, the image of it is an element of $\Upsilon^{\diamond}_{sta}$:

Lemma 4. *Assume $R \in \Xi$ is a preorder and a simulation relation. Then, $H(R) \in \Upsilon^{\diamond}_{sta}$.*

Let $\Xi^{\precsim} \subseteq \Xi$ be the set consisting of $R \in \Xi$ which is a preorder and a simulation relation. We show that the function obtained from H with restricted domain $\Xi^{\precsim}$ and co-domain $\Upsilon^{\diamond}_{sta}$, is bijective.

Lemma 5. *Let the function $h : \Xi^{\precsim} \to \Upsilon^{\diamond}_{sta}$ defined by: $h(R) = H(R)$ if $R \in \Xi^{\precsim}$. Then, h is bijective.*

Recall that $\precsim$ is a preorder, and is the largest simulation relation. We use $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$ to denote the partition pair $h(\precsim)$. Thus, $\precsim$ can be obtained via computing $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$. In the following lemma we show that $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$ is the unique, maximal element of $\Upsilon^\diamond_{sta}$:

Theorem 2 (Unique, Maximal Element). *The partition pair $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$ is the unique, maximal element of $\Upsilon^\diamond_{sta}$.*

Thus, to determine the simulation preorder $\precsim$, it is sufficient to compute the partition pair $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$. As in [6] we refer to it as the generalised coarsest partition problem (GCPP).

5 Solving the GCPP

Let $\mathcal{M} = (S, s_0, Act, \mathbf{P}, L)$ be a PA. In this section we propose an algorithm for solving the GCPP, i.e., computing the partition pair $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$ based on the partition refinement strategy. The idea is that we start with the partition pair $\langle \Sigma_0, \Gamma_0 \rangle$ which is coarser than $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$, and refine it with respect to the stability conditions. The Algorithm SimQuo is presented in Algorithm 1. As an initial partition pair we take $\Sigma_0 = \{\{s' \in S \mid L(s) = L(s') \wedge Act(s) = Act(s')\} \mid s \in S\}$. Intuitively, states with the same set of labels and enabled actions are put in the the same initial block. By construction Σ_0 is consistent with respect to L. The initial partition relation is defined by: $\Gamma_0 = \{(B, B') \in \Sigma_0 \times \Sigma_0 \mid L(B) = L(B') \wedge Act(B) \subseteq Act(B')\}$. Obviously, Γ_0 respects the labelling function L. It is easy to see that for partition pair $\langle \Sigma_i, \Gamma_i \rangle$ which is finer than $\langle \Sigma_0, \Gamma_0 \rangle$, Σ_i is consistent with respect to L, and Γ_i respects L as well.

In lines 5-15 of the algorithm, a finite sequence of partition pairs $\langle \Sigma_i, \Gamma_i \rangle$ with $i = 0, 1, \ldots, l$ is generated. We will show that it satisfies the following properties:

- Γ_i is acyclic for $i = 0, 1, \ldots, l$,
- $\langle \Sigma_i, \Gamma_i \rangle$ is coarser than $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$ for $i = 0, 1, \ldots, l$,
- $\langle \Sigma_{i+1}, \Gamma_{i+1} \rangle$ is finer than $\langle \Sigma_i, \Gamma_i \rangle$ for $i = 0, 1, \ldots, l-1$,
- $\langle \Sigma_l, \Gamma_l \rangle = \langle \Sigma^\diamond, \Gamma^\diamond \rangle$.

The core task consists of how to refine the partition pair $\langle \Sigma_i, \Gamma_i \rangle$ satisfying the above conditions.

In the non-probabilistic setting, a space-efficient algorithm [6] is proposed for a directed graph $G = (V, E)$. A refinement operator[1] was used to generate the partition pair $\langle \Sigma_{i+1}, \Gamma_{i+1} \rangle$ from $\langle \Sigma_i, \Gamma_i \rangle$ satisfying all of the properties mentioned above. The refinement of blocks works as follows. For each block $B \in \Sigma_i$ let $E^{-1}(B) = \{s \in V \mid \exists s' \in B.(s, s') \in E\}$ denote the set of predecessors of states in B. Then, using B' as a splitter, B is split into two part: $B_1 = B \cap E^{-1}(B')$ and $B_2 = B \setminus B_1$. The predecessor based method for splitting blocks, however, can

[1] The refinement operator must guarantee that the refined partition relation Γ_{i+1} must be acyclic. Recently, van Glabbeek and Ploeger [14] have shown that the operator in [6] was flawed, and provided a non-trivial fix for the operator.

Algorithm 1. SimQuo($\mathcal{M}$): Quotient algorithm to decide $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$ over $\mathcal{M}$

1: $i \leftarrow 0$
2: $\Sigma_0 = \{\{s' \in S \mid L(s) = L(s') \wedge Act(s) = Act(s')\} \mid s \in S\}$
3: $\Gamma_0 \leftarrow \{(B, B') \in \Sigma_0 \times \Sigma_0 \mid L(B) = L(B') \wedge Act(B) \subseteq Act(B')\}$
4: **repeat**
5: $\quad \Sigma_{i+1} \leftarrow \emptyset$, $\Gamma_{i+1} \leftarrow \emptyset$
6: $\quad$ **for all** $B \in \Sigma_i$ **do**
7: $\quad\quad \Sigma_{i+1} \leftarrow \Sigma_{i+1} \cup \text{Split}(B, \Sigma_i)$
8: $\quad \Gamma_{i+1} \leftarrow \{(Q, Q') \in \Sigma_{i+1} \times \Sigma_{i+1} \mid Par_{\Sigma_i}(Q) \neq Par_{\Sigma_i}(Q') \wedge (Par_{\Sigma_i}(Q), Par_{\Sigma_i}(Q')) \in \Gamma_i$ or $Par_{\Sigma_i}(Q) = Par_{\Sigma_i}(Q') \wedge Reach(Q, Q')\}$
9: $\quad$ Construct the $\exists$-quotient automaton $\exists\mathcal{M}/_{\Sigma_{i+1}}$
10: $\quad$ **repeat**
11: $\quad\quad$ **for all** $(Q, Q') \in \Gamma_{i+1}$ **do**
12: $\quad\quad\quad$ **if not** $(\forall Q \overset{\alpha}{\rightarrowtail} \pi_{\Sigma_{i+1}} \Rightarrow \exists Q' \overset{\alpha}{\rightarrow} \pi'_{\Sigma_{i+1}} \wedge \pi_{\Sigma_{i+1}} \sqsubseteq_{\Gamma_{i+1}} \pi'_{\Sigma_{i+1}})$ **then**
13: $\quad\quad\quad\quad \Gamma_{i+1} \leftarrow \Gamma_{i+1} \setminus \{(Q, Q')\}$
14: $\quad$ **until** Γ_{i+1} does not change
15: $\quad i++$
16: **until** $\langle \Sigma_{i+1}, \Gamma_{i+1} \rangle = \langle \Sigma_i, \Gamma_i \rangle$

not be applied to the probabilistic setting. The reason is that in PAs states have successor distributions instead of a single successor state. Moreover, the checking of the correspondence between distributions used for simulation involves weight functions which require additional attention. In the following we first propose a graph based analysis to refine the partition (lines 5–7). Then, we discuss how to refine the partition relation (lines 8–15).

Refinement of the Partition. Consider the partition pair $\langle \Sigma_i, \Gamma_i \rangle \in \Upsilon$ with $\langle \Sigma^\diamond, \Gamma^\diamond \rangle \ltimes \langle \Sigma_i, \Gamma_i \rangle$. The refinement operator Split consists of finding a finer partition Σ_{i+1} which is stable with respect to $\langle \Sigma_i, \Gamma_i^* \rangle$. For $B \in \Sigma_i$, $\text{Split}(B, \Sigma_i) = \{Q_1, \ldots, Q_k\}$ is a partition over B such that for all Q_i it should hold: if $Q_i \overset{\alpha}{\rightarrow} \pi_{\Sigma_i}$, there exists $Q_i \overset{\alpha}{\rightarrowtail} \pi'_{\Sigma_i}$ such that $\pi_{\Sigma_i} \sqsubseteq_{\Gamma_i^*} \pi'_{\Sigma_i}$ (cf. Definition 10). To construct this partition, we start with the following partition of B: $V_B = \{\{s' \in S \mid \forall \alpha \in Act(s).\ Steps_{\Sigma_i,\alpha}(s) = Steps_{\Sigma_i,\alpha}(s')\} \mid s \in S\}$. Note that V_B is finer than the partition for B we are searching for. We construct now a graph $G_B = (V_B, E_B)$ for the block B, in which for $Q, Q' \in V_B$, we add the edge $(Q, Q') \in E_B$ if the following condition holds:

$$\forall \pi_{\Sigma_i} \in Steps_{\Sigma_i,\alpha}(Q).\ \exists \pi'_{\Sigma_i} \in Steps_{\Sigma_i,\alpha}(Q').\ \pi_{\Sigma_i} \sqsubseteq_{\Gamma_i} \pi'_{\Sigma_i} \tag{1}$$

Note the condition $\pi_{\Sigma_i} \sqsubseteq_{\Gamma_i} \pi'_{\Sigma_i}$ could be checked via maximum flow computations [1]. We obtain the partition $\text{Split}(B, \Sigma_i)$ by constructing the *maximal strongly connected components* (SCCs) of G_B. Let $\text{Split}(B, \Sigma_i)$ denote the partition for B obtained by contracting the SCCs of G_B: $\text{Split}(B, \Sigma_i) = \{\cup_{X \in C} X \mid C \text{ is an SCC of } G_B\}$. Moreover, as in Algorithm SimQuo, let $\Sigma_{i+1} = \cup_{B \in \Sigma_i} \text{Split}(B, \Sigma_i)$. The following lemma shows that the obtained partition Σ_{i+1} is coarser than $\Sigma^\diamond$.

Lemma 6. *For all $i \geq 0$, Σ_{i+1} is finer than Σ_i, and $\Sigma^\diamond$ is finer than Σ_i.*

The following lemma shows that, for acyclic Γ_i, the partition Σ_{i+1} is stable with respect to $\langle \Sigma_i, \Gamma_i^* \rangle$:

Lemma 7. *Assume that Γ_i is a acyclic. For all $i \geq 0$, Σ_{i+1} is stable with respect to $\langle \Sigma_i, \Gamma_i^* \rangle$.*

Refinement of the Partition Relations. Similar to the refinement of partitions, at the end of iteration i, we want to get the partition relation Γ_{i+1} which is finer than Γ_i, but still coarser than $\Gamma^\diamond$. At line 8, the partition relation Γ_{i+1} is initialised such that it contains (Q, Q') if

- either Q, Q' have different parent blocks $B \neq B'$ with $B = Par_{\Sigma_i}(Q)$ and $B' = Par_{\Sigma_i}(Q')$ such that $(B, B') \in \Gamma_i$ holds,
- or they have same parent block B and the SCC for Q' can be reached by the SCC for Q in G_B. This constraint is abbreviated by $Reach(Q, Q')$ (line 8).

To get a coarser partition relation, we want to remove from Γ_{i+1} those pairs (B, B') satisfying the condition: no state in B can be simulated by any state in B'. Conversely, we want to keep those pairs (B, B') satisfying the condition that there exists at least a state in B which can be simulated by an arbitrary state in B'. This condition, however, depends on the concrete transitions of state $s \in B$. To be able to work completely on the quotient automaton $\exists\mathcal{M}/_{\Sigma_{i+1}}$, we consider the weakness of the above condition:

$$\forall B \overset{\alpha}{\rightarrowtail} \pi_{\Sigma_{i+1}} \Rightarrow \exists B' \overset{\alpha}{\rightarrow} \pi'_{\Sigma_{i+1}} \wedge \pi_{\Sigma_{i+1}} \sqsubseteq_{\Gamma_{i+1}} \pi'_{\Sigma_{i+1}} \tag{2}$$

Again, the condition $\pi_{\Sigma_{i+1}} \sqsubseteq_{\Gamma_{i+1}} \pi'_{\Sigma_{i+1}}$ could be checked via maximum flow computations [1]. Note the similarity to Condition 1: we consider only transitions of the form $B \overset{\alpha}{\rightarrowtail} \pi_{\Sigma_{i+1}}$ from B (line 12 in SimQuo).

Lemma 8. *For all $i \geq 0$, Γ_{i+1} is finer than Γ_i, and $\Gamma^\diamond$ is finer than Γ_i.*

Correctness. In this section we show the correctness of the algorithm SimQuo. By Lemmata 6 and 8, we see that the partition pair $\langle \Sigma_{i+1}, \Gamma_{i+1} \rangle$ obtained in the algorithm is finer than $\langle \Sigma_i, \Gamma_i \rangle$, and coarser than $\langle \Sigma^\diamond, \Gamma^\diamond \rangle$. The following lemma shows that the partition relation Γ_i is acyclic:

Lemma 9 (Acyclicity). *For all $i \geq 0$, the partition relation Γ_i is acyclic.*

Proof. We prove by induction on i. In the first iteration the statement holds: since the inclusion relation $\subseteq$ is transitive, no cycles except self loop exist in Γ_0. Now consider iteration i. By induction hypothesis assume that the partition relation Γ_i at the beginning of iteration i is acyclic. We shall show that Γ_{i+1} is acyclic until the end of i-te iteration. Consider the initial value of Γ_{i+1} at line 8 at iteration i. Note at this position we may still assume that Γ_i is acyclic by induction hypothesis. This implies that during the initialisation of Γ_{i+1} only sub-blocks from some same parent block $B \in \Sigma_i$ can form cycles of length $n > 1$. Assume such a cycle is formed from B: $Q_1 \Gamma_{i+1} Q_2 \Gamma_{i+1} \ldots, Q_n \Gamma_{i+1} Q_1 \ldots$ with $n > 1$. Since $Q_1 \Gamma_{i+1} Q_2$ implies that $Reach(Q_1, Q_2)$ and the reachability is

transitive, we get that $Q_1, \ldots, Q_n$ must belong to the same SCC in G_B which is a contradiction. Thus, Γ_{i+1} is acyclic after initialisation. Since afterwards pairs will only be removed from Γ_{i+1}, it remains acyclic.

Theorem 3 (Correctness). *Assume that* SimQuo *terminates at iteration l, then, $\langle \Sigma^\diamond, \Gamma^\diamond \rangle = \langle \Sigma_l, \Gamma_l \rangle$.*

Proof. By termination we have that $\langle \Sigma_l, \Gamma_l \rangle = \langle \Sigma_{l+1}, \Gamma_{l+1} \rangle$. By Lemma 9 the partition relation Γ_{l+1} is acyclic. Applying Lemma 7 we have that Σ_{l+1} is stable with respect to $\langle \Sigma_l, \Gamma_l^* \rangle$ which implies that Σ_l is stable with respect to $\langle \Sigma_l, \Gamma_l^* \rangle$. We prove first that the partition pair $\langle \Sigma_l, \Gamma_l^* \rangle$ is stable. Let $(B, B') \in \Gamma_l^*$, and $B \xrightarrow{\alpha} \pi_1$ with $\pi_1 \in \mathit{Dist}(\Sigma_l)$. Since Σ_l is stable with respect to $\langle \Sigma_l, \Gamma_l^* \rangle$, there must exist $\pi_1' \in \mathit{Dist}(\Sigma_l)$ with $B \overset{\alpha}{\rightarrowtail} \pi_1'$ such that $\pi_1 \sqsubseteq_{\Gamma_l^*} \pi_1'$. Since $(B, B') \in \Gamma_l^*$, there is a sequence $B_1, \ldots, B_n$ such that $B_1 \Gamma_l B_2 \Gamma_l \ldots B_n$ with $B_1 = B$ and $B_n = B'$ and $n \geq 2$. Σ_l is stable with respect to $\langle \Sigma_l, \Gamma_l^* \rangle$ implies that the block B_i is stable with respect to $\langle \Sigma_l, \Gamma_l^* \rangle$ for all $i = 1, \ldots, n$. Moreover, pairs in Γ_l satisfy Condition 2. Thus there exists distributions $\pi_i, \pi_i' \in \mathit{Dist}(\Sigma_l)$ for $i = 1, \ldots, n$ and such that it holds $B_i \xrightarrow{\alpha} \pi_i$, $B_i \overset{\alpha}{\rightarrowtail} \pi_i'$, and: $\pi_1 \sqsubseteq_{\Gamma_l^*} \pi_1' \sqsubseteq_{\Gamma_l} \pi_2 \sqsubseteq_{\Gamma_l^*} \pi_2' \sqsubseteq_{\Gamma_l} \ldots \pi_n \sqsubseteq_{\Gamma_l^*} \pi_n'$. Thus we have that $\pi_1 \sqsubseteq_{\Gamma_l^*} \pi_n'$ which implies that the partition pair $\langle \Sigma_l, \Gamma_l^* \rangle$ is stable. By Lemma 2 we have that $\langle \Sigma_l, \Gamma_l^* \rangle \ltimes \langle \Sigma^\diamond, \Gamma^\diamond \rangle$. By Lemmata 6 and 8 we have that $\langle \Sigma^\diamond, \Gamma^\diamond \rangle \ltimes \langle \Sigma_l, \Gamma_l \rangle$. Hence, $\langle \Sigma^\diamond, \Gamma^\diamond \rangle = \langle \Sigma_l, \Gamma_l \rangle$.

Complexity. The following lemma shows that the number of iterations of the algorithm SimQuo is linear in $|\Sigma^\diamond|$:

Lemma 10. *Assume that* SimQuo *terminates at iteration l, then, $l \in \mathcal{O}(|\Sigma^\diamond|)$.*

For the complexity analysis, we introduce some notations. As before, given $\mathcal{M} = (S, s_0, \mathit{Act}, \mathbf{P}, L)$, we let n, m denote the number of states and transitions respectively. We let $\Sigma_\sim$ denote the partition induced by the bisimulation relation $\sim$, and let $n_\sim$ and $m_\sim$ denote the number of states and transitions of the $\exists$-quotient[2] automaton $\exists\mathcal{M}/_{\Sigma_\sim}$. Let $n_\diamond$ and $m_\diamond$ denote the number of states and transitions of the quotient automaton $\exists\mathcal{M}/_{\Sigma^\diamond}$.

Theorem 4 (Complexity). *Given a PA $\mathcal{M}$, the algorithm* SimQuo *has time complexity $\mathcal{O}(mn_\diamond + m_\diamond^2 n_\diamond^4 + m_\sim^2 n_\diamond^2)$, and space complexity $\mathcal{O}(n_\diamond^2 + n \log n_\diamond)$.*

The space complexity can be considered optimal: the $\mathcal{O}(n_\diamond^2)$ part is needed to save the partition relations, and the $\mathcal{O}(n \log n_\diamond)$ part is needed to save to which simulation equivalence class a state belongs. The worst case time complexity in each iteration is in the order of $\mathcal{O}(m + m_\diamond^2 n_\diamond^3 + m_\sim^2 n_\diamond^3)$. Together with Lemma 10 we get a rather excessive time complexity. We consider an iteration i of the algorithm SimQuo. In the inside repeat loop of this iteration, the weight function condition $\pi_{\Sigma_{i+1}} \sqsubseteq_{\Gamma_{i+1}} \pi'_{\Sigma_{i+1}}$ (see Condition 2) can be checked via solving a

[2] In fact, the $\exists$-quotient automaton and the $\forall$-quotient automaton with respect to the bisimulation relation $\sim$ coincide.

maximum flow problem over a network constructed out of $\pi_{\Sigma_{i+1}}$, $\pi'_{\Sigma_{i+1}}$ and Γ_{i+1}. Observe that the networks on which the maximum flows are calculated in the inside repeat loop are very similar. Similar to algorithms in PAs [16], we could apply parametric maximum flow techniques (PMFs) to improve the time complexity: instead of recompute the maximum we could reuse the maximum computed in the last iteration. The penalty is that more memory is needed due to the need to store the networks and flows of it across iterations.

Theorem 5 (Complexity with PMFs). *Using PMFs, the algorithm* SimQuo *has time complexity* $\mathcal{O}(mn_\diamond + m^2_\sim n^2_\diamond)$*, and space complexity is* $\mathcal{O}(m^2_\diamond + n \log n_\diamond)$.

6 Experimental Results

In this section, we evaluate our new partition refinement based algorithm. Depending whether PMFs are used in the algorithm, we have implemented both the space-efficient and time-efficient variants of the partition refinement based algorithm. We compare the results to previous algorithms in [1,16]. All experiments were run on a Linux machine with an AMD Athlon(tm) XP 2600+ processor at 2 GHz equipped with 2GB of RAM.

Dining Cryptographers. Consider the dining cryptographer models taken from the PRISM web-site. In [2], various optimisations are proposed for computing probabilistic simulation. We take the most space efficient configuration (0000) and refer to it as the *Original* algorithm in the sequel. Note that other configurations use more memory, and are at most faster by a factor of two, thus are not considered here. We compare it to our new partition refinement based algorithm: the configuration QuoPMF for the algorithm using PMFs and the configuration Quotient for the algorithm without using PMFs.

In Table 1 experiments are shown: in the upper part only one state label is considered, in the middle part uniform distribution of two different labels is considered, in the lower part we have uniform distribution of three different labels. For 6 cryptographers and one or two labels, the configuration Original runs out of memory; this is denoted by –. The number of the simulation equivalence classes is given in row #blocks, and the number of iterations of the refinement loops for the configurations Quotient and QuoPMF is given in row #refinement.

As expected, in the configuration Original the memory is indeed the bottleneck, while the partition refinement based algorithm uses significant less memory. More surprisingly is that partition refinement based algorithm often requires orders of magnitude less time, especially for small number of labels. The reason is that for this case study the simulation quotient automaton has much less states than the original automaton. Moreover, in the quotient automaton, most of the transitions fall into the same lifted distributions, thus making the maximum flow computation cheaper. Another observation is that the number of different labels affect the performance of all of the configurations, but in a different way. For the configuration Original more labels indicate that the initial relation is smaller thus always less time and memory are needed. For both Quotient and QuoPMF more labels give a finer initial partition, which means also a large quotient automaton during

Table 1. Time and memory used for Dining Cryptographers

Cryptographers	3	4	5	6	3	4	5	6
States	381	2166	11851	63064	381	2166	11851	63064
Transitions	780	5725	38778	246827	780	5725	38778	246827
	Time (s)				Memory (MB)			
Original	0.52	20.36	987.40	–	0.95	27.41	763.09	–
Quotient	0.03	0.76	19.52	533.40	0.02	0.11	0.71	4.35
QuoPMF	0.03	0.73	18.93	528.00	0.02	0.14	0.89	5.25
#blocks	10	24	54	116				
#refinement	3	3	3	3				
Original	0.13	4.67	266.04	–	0.21	4.68	104.46	–
Quotient	0.05	0.93	18.53	394.80	0.02	0.12	0.93	7.07
QuoPMF	0.05	0.96	19.46	420.60	0.02	0.21	2.42	26.02
#blocks	63	247	955	3377				
#refinement	4	4	4	4				
Original	0.07	2.42	150.74	13649.30	0.14	2.69	58.92	1414.57
Quotient	0.06	2.31	60.01	1185.16	0.02	0.18	2.32	22.67
QuoPMF	0.07	3.04	81.14	1536.78	0.03	0.41	10.75	124.53
#blocks	96	554	2597	8766				
#refinement	3	4	4	5				

the refinement loops. For this example the running time for one or two labels are almost the same, whereas with three labels more time is needed.

It is notable that the QuoPMF configuration does not perform well at all, even though it has better theoretical complexity in time. This observation is the same as we have observed in [2]: the corner cases (number of iterations in the inside repeat-loop is bounded by $n_\diamond^2$) which blow up the worst case complexity are rare in practice.

Self Stabilising Algorithm. We now consider the self stabilising algorithm due to Israeli and Jalfon, also taken from the PRISM web-site. As the previous case study, in the upper, middle and lower part of the table we have one, two and three different uniformly distributed labels respectively. For 13 processes and one label, the configuration QuoPMF runs out of memory which is denoted by –. For this case study, we observe that the simulation quotient automaton has almost the same number of states as the original one. Thus, Original is the fastest configuration. Another observation is that the configuration Quotient needs almost the same amount of memory for three different number of labels. Recall that the space complexity of the configuration Quotient is $\mathcal{O}(n_\diamond^2 + n \log n_\diamond)$. In this case study the number of blocks differs only slightly for different number of labels, thus almost the same amount of memory is needed for this configuration.

Random Models. Most of the real models have a sparse structure: the number of successor distributions and the size of each distribution are small. Now we consider randomly generated PAs in which we can also observe how the algorithms

Table 2. Time and memory used for the self stabilising algorithm

Processes	10	11	12	13	10	11	12	13
States	1023	2047	4095	8191	1023	2047	4095	8191
Transitions	8960	19712	43008	93184	8960	19712	43008	93184
	Time (s)				Memory (MB)			
Original	11.35	53.66	259.18	1095.96	5.88	20.10	91.26	362.11
Quotient	20.25	138.60	470.84	2440.83	0.36	1.24	4.50	17.04
QuoPMF	28.17	177.54	655.09	–	93.40	375.47	1747.35	–
#blocks	974	1987	4024	8107				
#refinement	6	6	7	7				
Original	1.73	8.68	37.63	199.31	0.92	3.34	12.42	47.25
Quotient	10.60	52.60	234.96	1248.30	0.38	1.29	4.63	17.35
QuoPMF	14.57	73.06	325.82	1704.87	17.93	80.14	338.45	1379.45
#blocks	1019	2042	4090	8185				
#refinement	5	6	6	7				
Original	0.61	2.47	13.56	66.62	0.47	1.42	5.28	18.38
Quotient	10.36	39.02	260.09	900.99	0.38	1.29	4.62	17.35
QuoPMF	14.29	54.34	360.63	1235.27	2.24	11.97	28.93	142.68
#blocks	1015	2042	4085	8185				
#refinement	6	5	8	6				

Table 3. Random models with various maximal distribution size D

D	5	7	9	11	13	15	17	19
Transitions	1927	2717	3121	3818	4040	4711	5704	6389
	Time (s)							
Original	0.50	1.10	1.80	3.19	3.76	6.04	10.26	14.12
Quotient	0.58	0.56	0.56	0.60	0.63	0.64	0.72	0.78
QuoPMF	0.54	0.54	0.52	0.59	0.60	0.60	0.70	0.74
#refinement	4	3	3	3	3	3	3	3
	Memory (kB)							
Original	138.23	137.58	108.18	132.85	115.10	131.88	145.19	144.30
Quotient	37.89	47.69	52.91	61.44	64.68	72.58	84.99	93.22
QuoPMF	263.77	179.51	128.60	144.11	107.94	83.46	110.10	106.02

behave for dense models. We consider random model with 200 states, in which there are two actions $|Act| = 2$, the size of each α-successor distribution in the model is uniform distributed between $\{2, \ldots, D\}$, and the number of successor distributions for each state is uniform distributed between $\{1, \ldots, MS\}$. Only one state label is considered.

In Table 3 we set $MS = 2$ and consider various values of D. Because of the large distribution size, in all of these random models the simulation quotient automaton is the same as the corresponding original automaton, thus there is no reduction at all. Even in this extreme case, the partition refinement based methods reduce the memory by approximately 30%. Because of the large size of

Table 4. Random models with various maximal number of successor distributions *MS*

MS	10	15	20	25	30	35	40	45
Transitions	3732	5283	7432	9250	11217	12659	13800	16170
	Time (s)							
Original	2.62	6.40	25.49	26.18	29.92	18.63	23.35	13.30
Quotient	1.15	2.97	6.82	4.88	4.44	2.83	4.67	2.45
QuoPMF	1.26	3.56	7.68	4.98	4.51	2.82	4.74	2.52
#blocks	200	200	200	13	22	9	11	5
#refinement	4	5	9	6	4	3	4	2
	Memory (kB)							
Original	348.79	437.73	501.16	567.91	575.46	628.32	633.17	670.90
Quotient	61.07	81.00	108.54	121.71	147.15	165.33	180.14	210.29
QuoPMF	1063.00	1663.16	2831.99	149.80	184.65	171.88	190.35	211.19

distributions, the corresponding maximum flow computations become more expensive for the configuration Original. In the partition refinement based approach the maximum flow computations are carried in the quotient automaton in each iteration, which saves considerable time. Thus the partition refinement based methods are faster, and scale much better than the configuration Original. Comparing with the configuration Quotient, the parametric maximum flow based method (configuration QuoPMF) uses more memory, and has only negligible time advantages. In Table 4 we fix the maximal size of distribution to $D = 5$, and consider various values of *MS*. With the increase of *MS*, it is more probable that states are simulation equivalent, which means also that the number of blocks tends to be smaller for large *MS*. Also for this kind of dense models, we observe that significant time and space advantages are achieved. Again, the PMF-based method does not perform better in time, and uses more memory.

7 Conclusions

In this paper we proposed a partition refinement based algorithm for deciding simulation preorders. The space complexity of our algorithm is as good as the best one for the non-probabilistic setting, which is a special case of this setting. We discussed how to reduce the time complexity further by exploiting parametric maximum flow algorithms. Our implementation of the space-efficient and time-efficient variants of the algorithm has given experimental evidence, comparing to the original algorithm, not only the space-efficiency is improved drastically. Often the computation time is decreased by orders of magnitude.

Acknowledgements. Thanks to Jonathan Bogdoll for helping us with the implementation, to Holger Hermanns, Joost-Pieter Katoen, and Frits W. Vaandrager for providing many valuable pointers to the literature.

References

1. Baier, C., Engelen, B., Majster-Cederbaum, M.E.: Deciding bisimilarity and similarity for probabilistic processes. J. Comput. Syst. Sci. 60(1), 187–231 (2000)
2. Bogdoll, J., Hermanns, H., Zhang, L.: An experimental evaluation of probabilistic simulation. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 37–52. Springer, Heidelberg (2008)
3. Bustan, D., Grumberg, O.: Simulation based minimization. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 255–270. Springer, Heidelberg (2000)
4. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)
5. Gallo, G., Grigoriadis, M., Tarjan, R.: A fast parametric maximum flow algorithm and applications. SIAM J. Comput. 18(1), 30–55 (1989)
6. Gentilini, R., Piazza, C., Policriti, A.: From bisimulation to simulation: Coarsest partition problems. J. Autom. Reasoning 31(1), 73–103 (2003)
7. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: FOCS, pp. 453–462 (1995)
8. Jonsson, B., Larsen, K.: Specification and refinement of probabilistic processes. In: LICS, pp. 266–277 (1991)
9. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
10. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
11. Ranzato, F., Tapparo, F.: A new efficient simulation equivalence algorithm. In: LICS, pp. 171–180 (2007)
12. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. Nord. J. Comput. 2(2), 250–273 (1995)
13. Tan, L., Cleaveland, R.: Simulation revisited. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 480–495. Springer, Heidelberg (2001)
14. van Glabbeek, R.J., Ploeger, B.: Correcting a space-efficient simulation algorithm. In: CAV 2008 (to appear, 2008)
15. Zhang, L.: Decision Algorithms for Probabilistic Simulations. PhD thesis, Universitätm des Saarlandes (2008)
16. Zhang, L., Hermanns, H.: Deciding simulations on probabilistic automata. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 207–222. Springer, Heidelberg (2007)
17. Zhang, L., Hermanns, H., Eisenbrand, F., Jansen, D.N.: Flow faster: Efficient decision algorithms for probabilistic simulations. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 155–169. Springer, Heidelberg (2007)

Least Upper Bounds for Probability Measures and Their Applications to Abstractions

Rohit Chadha[1,⋆], Mahesh Viswanathan[1,⋆⋆], and Ramesh Viswanathan[2]

[1] University of Illinois at Urbana-Champaign
{rch,vmahesh}@uiuc.edu
[2] Bell Laboratories
rv@research.bell-labs.com

Abstract. Abstraction is a key technique to combat the state space explosion problem in model checking probabilistic systems. In this paper we present new ways to abstract Discrete Time Markov Chains (DTMCs), Markov Decision Processes (MDPs), and Continuous Time Markov Chains (CTMCs). The main advantage of our abstractions is that they result in abstract models that are *purely probabilistic*, which maybe more amenable to automatic analysis than models with both nondeterministic and probabilistic steps that typically arise from previously known abstraction techniques. A key technical tool, developed in this paper, is the construction of least upper bounds for any collection of probability measures. This upper bound construction may be of independent interest that could be useful in the abstract interpretation and static analysis of probabilistic programs.

1 Introduction

Abstraction is an important technique to combat *state space explosion*, wherein a smaller, abstract model that conservatively approximates the behaviors of the original (concrete) system is verified/model checked. Typically abstractions are constructed on the basis of an equivalence relation (of finite index) on the set of (concrete) states of the system. The abstract model has as states the equivalence classes (*i.e.*, it collapses all equivalent states into one), and each abstract state has transitions corresponding to the transitions of each of the concrete states in the equivalence class. Thus, the abstract model has both nondeterministic and (if the concrete system is probabilistic) probabilistic behavior.

In this paper, we present new methods to abstract probabilistic systems modeled by Discrete Time Markov Chains (DTMC), Markov Decision Processes (MDP), and Continuous Time Markov Chains (CTMC). The main feature of our constructions is that the resulting abstract models are *purely probabilistic* in that they do not have any nondeterministic choices. Since analyzing models that have both nondeterministic and probabilistic behavior is more challenging than

⋆ Supported partially by NSF CCF 04-29639.
⋆⋆ Supported partially by NSF CCF 04-48178.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 264–278, 2008.

analyzing models that are purely probabilistic, we believe that this may make our abstractions more amenable to automated analysis; the comparative tractability of model-checking systems without non-determinism is further detailed later in this section.

Starting from the work of Saheb-Djahromi [19], and further developed by Jones [11], orders on measures on special spaces (Borel sets generated by Scott open sets of a cpo) have been used in defining the semantics of probabilistic programs. Ordering between probability measures also play a central role in defining the notion of simulation for probabilistic systems. For a probabilistic model, a transition can be viewed as specifying a probability measure on successor states. One transition then simulates another if the probability measures they specify are related by the ordering on measures. In this manner, simulation and bisimulation relations were first defined for DTMCs and MDPs [12], and subsequently extended to CTMCs [3]. Therefore, in all these settings, a set of transitions is abstracted by a transition if it is an upper bound for the probability measures specified by the set of transitions being abstracted.

The key technical tool that we develop in this paper is a new construction of least upper bounds for arbitrary sets of probability measures. We show that in general, measures (even over simple finite spaces) do not have least upper bounds. We therefore look for a class of measurable spaces for which the existence of least upper bounds is guaranteed for arbitrary sets of measures. Since the ordering relation on measures is induced from the underlying partial order on the space over which the measures are considered, we identify conditions on the underlying partial order that are sufficient to prove the existence of least upper bounds — intuitively, these conditions correspond to requiring the Hasse diagram of the partial order to have a "tree-like" structure. Furthermore, we show that these conditions provide an exact characterization of the measurable spaces of our interest — for any partial order not satisfying these conditions, we can construct two measures that do not have a least upper bound. Finally, for this class of tree-like partial orders, we provide a natural construction that is proven to yield a well-defined measure that is a least upper bound.

These upper bound constructions are used to define abstractions as follows. As before, the abstract model is defined using an equivalence relation on the concrete states. The abstract states form a tree-like partial order with the minimal elements consisting of the equivalence classes of the given relation. The transition out of an abstract state is constructed as the least upper bound of the transitions from each of the concrete states it "abstracts". Since each upper bound is a single measure yielding a single outgoing transition, the resulting abstract model does not have any nondeterminism. This intuitive idea is presented and proved formally in the context of DTMCs, MDPs and CTMCs.

A few salient features of our abstract models bear highlighting. First, the fact that least upper bounds are used in the construction implies that for a particular equivalence relation on concrete states and partial order on the abstract states, the abstract model constructed is finer than (*i.e.*, can be simulated by) any purely probabilistic models that can serve as an abstraction. Thus, for verification

purposes, our model is the most precise purely probabilistic abstraction on a chosen state space. Second, the set of abstract states is not completely determined by the equivalence classes of the relation on concrete states; there is freedom in the choice of states that are above the equivalence classes in the partial order. However, for any such choice that respects the "tree-like" requirement on the underlying partial order, the resulting model will be exponentially smaller than the existing proposals of [8,13]. Furthermore, we show that there are instances where we can get more precise results than the abstraction schemes of [8,13] while using significantly fewer states (see Example 4). Third, the abstract models we construct are purely probabilistic which makes model checking easier. Additionally, these abstractions can potentially be verified using statistical techniques which do not work when there is nondeterminism [24,23,21]. Finally, CTMC models with nondeterminism, called CTMDP, are known to be difficult to analyze [2]. Specifically, the measure of time-bounded reachability can only be computed approximately through an iterative process; therefore, there is only an approximate algorithm for model-checking CTMDPs against CSL. On the other hand, there is a theoretically exact solution to the corresponding model-checking problem for CTMCs by reduction to the first order theory of reals [1].

Related Work. Abstractions have been extensively studied in the context of probabilistic systems. General issues and definitions of good abstractions are presented in [12,9,10,17]. Specific proposals for families of abstract models include Markov Decision Processes [12,5,6], systems with interval ranges for transition probabilities [12,17,8,13], abstract interpretations [16], 2-player stochastic games [14], and expectation transformers [15]. Recently, theorem-prover based algorithms for constructing abstractions of probabilistic systems based on predicates have been presented [22]. All the above proposals construct models that exhibit both nondeterministic and probabilistic behavior. The abstraction method presented in this paper construct purely probabilistic models.

2 Least Upper Bounds for Probability Measures

This section presents our construction of least upper bounds for probability measures. Section 2.1 recalls the relevant definitions and results from measure theory. Section 2.2 presents the ordering relation on measures. Finally, Section 2.3 presents the least upper bound construction on measures. Due to space considerations, many of the proofs are deferred to [4] for the interested reader.

2.1 Measures

A **measurable space** (X, Σ) is a set X together with a family of subsets, Σ, of X, called a **σ-field** or **σ-algebra**, that contains $\emptyset$ and is closed under complementation and countable union. The members of a σ-field are called the **measurable subsets** of X. Examples of σ-fields are $\{\emptyset, X\}$ and $\mathcal{P}(X)$ (the powerset of X). We will sometimes abuse notation and refer to the measurable

space (X, Σ) by X or by Σ, when the σ-field or set, is clear from the context. The intersection of an arbitrary collection of σ-fields on a set X is again a σ-field, and so given any $B \subseteq \mathcal{P}(X)$ there is a least σ-field containing B, which is called the σ-field **generated** by B.

A **positive measure** μ on a measurable space (X, Σ) is a function from Σ to $[0, \infty]$ such that μ is **countably additive**, *i.e.*, if $\{A_i \mid i \in I\}$ is a countable family of pairwise disjoint measurable sets then $\mu(\bigcup_{i \in I} A_i) = \sum_{i \in I} \mu(A_i)$. In particular, if $I = \emptyset$, we have $\mu(\emptyset) = 0$. A measurable space equipped with a measure is called a **measure space**. The **total weight** of a measure μ on measurable space X is $\mu(X)$. A **probability measure** is a positive measure of total weight 1. We denote the collection of all probability measures on X by $\mathcal{M}_{=1}(X)$.

2.2 A Partial Order on Measures

In order to define an ordering on probability measures we need to consider measurable spaces that are equipped with an ordering relation. An *ordered measurable space* $(X, \Sigma, \sqsubseteq)$ is a set X equipped with a σ-field Σ and a preorder on X [1] $\sqsubseteq$ such that (X, Σ) is a measurable space. A (probability) measure on $(X, \Sigma, \sqsubseteq)$ is a (probability) measure on (X, Σ). Finally, recall that a set $U \subseteq X$ is *upward closed* if for every $x \in U$ and $y \in X$ with $x \sqsubseteq y$ we have that $y \in U$. The ordering relation on the underlying set is lifted to an ordering relation on probability measures as follows.

Definition 1. *Let $\mathcal{X} = (X, \Sigma, \sqsubseteq)$ be an ordered measurable space. For any probability measures μ, ν on $\mathcal{X}$, define $\mu \leq \nu$ iff for every upward closed set $U \in \Sigma$, $\mu(U) \leq \nu(U)$.*

Our definition of the ordering relation is formulated so as to be applicable to any general measurable space. For probability distributions over finite spaces, it is equivalent to a definition of *lifting of preorders to probability measures* using *weight functions* as considered in [12] for defining simulations. Indeed, Definition 1 can be seen to be identical to the presentation of the simulation relation in [7,20] where this equivalence has been observed as well.

Recall that a set $D \subseteq X$ is *downward closed* if for every $y \in D$ and $x \in X$ with $x \sqsubseteq y$ we have that $x \in D$. The ordering relation on probability measures can be dually cast in terms of downward closed sets which is useful in the proofs of our construction.

Proposition 1. *Let $\mathcal{X} = (X, \Sigma, \sqsubseteq)$ be an ordered measurable space. For any probability measures μ, ν on $\mathcal{X}$, we have that $\mu \leq \nu$ iff for every downward closed set $D \in \Sigma$, $\mu(D) \geq \nu(D)$.*

In general, Definition 1 yields a preorder that is not necessarily a partial order. We identify a special but broad class of ordered measurable spaces for which the

[1] Recall that preorder on a set X is a binary relation $\sqsubseteq \subseteq X \times X$ such that $\sqsubseteq$ is reflexive and transitive.

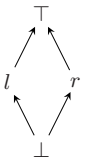


Fig. 1. Hasse Diagram of T. Arrows directed from smaller element to larger element.

ordering relation is a partial order. The spaces we consider are those which are generated by some collection of downward closed sets.

Definition 2. *An ordered measurable space* $(X, \Sigma, \sqsubseteq)$ *is* order-respecting *if there exists* $\mathcal{D} \subseteq \mathcal{P}(X)$ *such that every* $D \in \mathcal{D}$ *is downward closed (with respect to* $\sqsubseteq$*) and* Σ *is generated by* $\mathcal{D}$*.*

Example 1. For any finite set A, the space $(\mathcal{P}(A), \mathcal{P}(\mathcal{P}(A)), \subseteq)$ is order-respecting since it is generated by all the downward closed sets of $(\mathcal{P}(A), \subseteq)$. One special case of such a space that we will make use of in our examples is where $T = \mathcal{P}(\{0,1\})$ whose Hasse diagram is shown in Figure 1; we will denote the elements of T by $\bot = \emptyset$, $l = \{0\}$, $r = \{1\}$, and $\top = \{0,1\}$. Then $\mathbb{T} = (T, \mathcal{P}(T), \subseteq)$ is an order-respecting measurable space. Finally, for any cpo $(X, \sqsubseteq)$, the Borel measurable space $(X, \mathcal{B}(X), \sqsubseteq)$ is order-respecting since every Scott-closed set is downward closed.

Theorem 1. *For any ordered measurable space* $\mathcal{X} = (X, \Sigma, \sqsubseteq)$*, the relation* $\leq$ *is a preorder on* $\mathcal{M}_{=1}(\mathcal{X})$*. The relation* $\leq$ *is additionally a partial order (anti-symmetric) if* $\mathcal{X}$ *is order-respecting.*

Example 2. Recall the space $\mathbb{T} = (T, \mathcal{P}(T), \subseteq)$ defined in Example 1. Consider the probability measure λ, where l has probability 1, and all the rest have probability 0. Similarly, τ is the measure where $\top$ has probability 1, and the rest 0, and in ρ, r gets probability 1, and the others 0. Now one can easily see that $\lambda \leq \tau$ and $\rho \leq \tau$. However $\lambda \not\leq \rho$ and $\rho \not\leq \lambda$.

2.3 Construction of Least Upper Bounds

Least upper bound constructions for elements in a partial order play a crucial role in defining the semantics of languages as well as in abstract interpretation. As we shall show later in this paper, least upper bounds of probabilistic measures can also be used to define abstract models of probabilistic systems. Unfortunately, however, probability measures over arbitrary measurable spaces do not necessarily have least upper bounds; this is demonstrated in the following example.

Example 3. Consider the space $\mathbb{T}$ defined in Example 1. Let μ be the probability measure that assigns probability $\frac{1}{2}$ to $\bot$ and l, and 0 to everything else. Let ν be such that it assigns $\frac{1}{2}$ to $\bot$ and r, 0 to everything else. The measure τ that

assigns $\frac{1}{2}$ to $\top$ and $\bot$ is an upper bound of both μ and ν. In addition, ρ that assigns $\frac{1}{2}$ to l and r, and 0 to everything else, is also an upper bound. However τ and ρ are incomparable. Moreover, any lower bound of τ and ρ must assign a probability at least $\frac{1}{2}$ to $\bot$ and probability 0 to $\top$, and so cannot be an upper bound of μ and ν. Thus, μ and ν do not have a least upper bound.

We therefore identify a special class of ordered measurable spaces over which probability measures admit least upper bounds. Although our results apply to general measurable spaces, for ease of understanding, the main presentation here is restricted to finite spaces. For the rest of the section, fix an ordered measurable space $\mathcal{X} = (X, \mathcal{P}(X), \sqsubseteq)$, where $(X, \sqsubseteq)$ is a finite partial order. For any element $a \in X$, we use D_a to denote the downward-closed set $\{b \mid b \sqsubseteq a\}$. We begin by defining a *tree-like* partial order; intuitively, these are partial orders whose Hasse diagram resembles a tree (rooted at its greatest element).

Definition 3. *A partial order $(X, \sqsubseteq)$ is said to be* tree-like *if and only if (i) X has a greatest element $\top$, and (ii) for any two elements $a, b \in X$ if $D_a \cap D_b \neq \emptyset$ then either $D_a \subseteq D_b$ or $D_b \subseteq D_a$.*

We can show that over spaces whose underlying ordering is tree-like, any set of probability measures has a least upper bound. This construction is detailed in Theorem 2 and its proof below.

Theorem 2. *Let $\mathcal{X} = (X, \mathcal{P}(X), \sqsubseteq)$ be an ordered measurable space where $(X, \sqsubseteq)$ is tree-like. For any $\Gamma \subseteq \mathcal{M}_{=1}(\mathcal{X})$, there is a probability measure $\nabla(\Gamma)$ such that $\nabla(\Gamma)$ is the least upper bound of Γ.*

Proof. Recall that for a set $S \subseteq X$, its set of maximal elements maximal(S) is defined as $\{a \in S \mid \forall b \in S.\ a \sqsubseteq b \Rightarrow a = b\}$. For any downward closed set D, we have that $D = \cup_{a \in \text{maximal}(D)} D_a$. From condition (ii) of Definition 3, if a, b are two distinct maximal elements of a downward closed set D then $D_a \cap D_b = \emptyset$ and the sets comprising the union are pairwise disjoint. For any measure μ, we therefore have that $\mu(D) = \sum_{a \in \text{maximal}(D)} \mu(D_a)$ for any downward closed set D.

Define the function ν on downward closed subsets of X as follows. For a downward closed set of the form D_a, where $a \in X$, take $\nu(D_a) = \inf_{\mu \in \Gamma} \mu(D_a)$, and for any downward closed set D take $\nu(D) = \sum_{a \in \text{maximal}(D)} \nu(D_a)$. We will define the least upper bound measure $\nabla(\Gamma)$ by specifying its value pointwise on each element of X. Observe that for any $a \in X$, the set $D_a \setminus \{a\}$ is also downward closed. We therefore define $\nabla(\Gamma)(\{a\}) = \nu(D_a) - \nu(D_a \setminus \{a\})$, for any $a \in X$.

Observe that $\nu(D) \leq \inf_{\mu \in \Gamma} \mu(D)$. We therefore have that $\nabla(\Gamma)(\{a\}) \geq 0$. For any downward closed set D, we can see that $\nabla(\Gamma)(D) = \nu(D)$. Thus, $\nabla(\Gamma)(X) = \nabla(\Gamma)(D_\top) = \nu(D_\top) = \inf_{\mu \in \Gamma} \mu(D_\top) = 1$, and so $\nabla(\Gamma)$ is a probability measure on $\mathcal{X}$.

For any downward closed set D, we have that $\nabla(\Gamma)(D) = \nu(D)$ and $\nu(D) \leq \inf_{\mu \in \Gamma} \mu(D)$ which allows us to conclude that $\nabla(\Gamma)$ is an upper bound of Γ. Now consider any measure τ that is an upper bound of Γ. Then, $\tau(D) \leq \mu(D)$ for any measure $\mu \in \Gamma$ and all downward closed sets D. In particular, for any element $a \in$

$X, \tau(D_a) \leq \inf_{\mu \in \Gamma} \mu(D_a) = \nu(D_a) = \nabla(\Gamma)(D_a)$. Thus, for any downward closed set D, we have that $\tau(D) = \sum_{a \in \text{maximal}(D)} \tau(D_a) \leq \sum_{a \in \text{maximal}(D)} \nabla(\Gamma)(D_a) = \nabla(\Gamma)(D)$. Hence, $\nabla(\Gamma) \leq \tau$, which concludes the proof. □

We conclude this section, by showing that if we consider any ordered measurable space that is not tree-like, there are measures that do not have least upper bounds. Thus, the tree-like condition is an *exact*(necessary and sufficient) characterization of spaces that admit least upper bounds of arbitrary sets of probability measures.

Theorem 3. *Let $\mathcal{X} = (X, \mathcal{P}(X), \sqsubseteq)$ be an ordered measurable space, where $(X, \sqsubseteq)$ is a partial order that is* not *tree-like. Then there are probability measures μ and ν such that μ and ν do not have a least upper bound.*

Proof. First consider the case when X does not have a greatest element. Then there are two maximal elements, say a and b. Let μ be the measure that assigns measure 1 to a and 0 to everything else, and let ν be the measure that assigns 1 to b and 0 to everything else. Clearly, μ and ν do not have an upper bound.

Next consider the case when X does have a greatest element $\top$; the proof in this case essentially follows from generalizing Example 3. If $\mathcal{X}$ is a space as in the theorem then since $(X, \sqsubseteq)$ is not tree-like, there are two elements $a, b \in X$ such that $D_a \cap D_b \neq \emptyset$, $D_a \setminus D_b \neq \emptyset$, and $D_b \setminus D_a \neq \emptyset$. Let $c \in D_a \cap D_b$. Consider the measure μ to be such that $\mu(\{c\}) = \frac{1}{2}$, $\mu(\{a\}) = \frac{1}{2}$, and is equal to 0 on all other elements. Define the measure ν to be such that $\nu(\{c\}) = \frac{1}{2}$, $\nu(\{b\}) = \frac{1}{2}$, and is equal to 0 on all other elements. As in Example 3, we can show that μ and ν have two incomparable minimal upper bounds. □

Remark 1. All the results presented in the section can be extended to ordered measure spaces $\mathcal{X} = (X, \mathcal{P}(X), \sqsubseteq)$ when X is a countable set; see [4].

3 Abstracting DTMCs and MDPs

In this section we outline how our upper bound construction can be used to abstract MDPs and DTMCs using DTMCs. We begin by recalling the definitions of these models along with the notion of simulation and logic preservation in Section 3.1, before presenting our proposal in Section 3.2.

3.1 Preliminaries

We recall 3-valued PCTL and its discrete time models. In 3-valued logic, a formula can evaluate to either *true* ($\top$), *false* ($\bot$), or *indefinite* (?); let $\mathbb{B}_3 = \{\bot, ?, \top\}$. The formulas of PCTL are built up over a finite set of atomic propositions AP and are inductively defined as follows.

$$\varphi ::= \text{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{P}_{\bowtie p}(X\varphi) \mid \mathcal{P}_{\bowtie p}(\varphi \,\mathcal{U}\, \varphi)$$

where $a \in \text{AP}$, $\bowtie \in \{<, \leq, >, \geq\}$, and $p \in [0, 1]$.

The models of these formulas are interpreted over *Markov Decision Processes*, formally defined as follows. Let Q be a finite set of states and let $\mathcal{Q} = (Q, \mathcal{P}(Q))$

be a measure space. A Markov Decision Process (MDP) $\mathcal{M}$ is a tuple $(Q, \rightarrow, L)$, where $\rightarrow \subseteq Q \times \mathcal{M}_{=1}(\mathcal{Q})$ (non-empty and finite), and $L : (Q \times \mathrm{AP}) \rightarrow \mathbb{B}_3$ is a labeling function that assigns a value in $\mathbb{B}_3$ to each atomic proposition in each state. We will say $q \rightarrow \mu$ to mean $(q, \mu) \in \rightarrow$. A *Discrete Time Markov Chain* (DTMC) is an MDP with the restriction that for each state q there is exactly one probability measure μ such that $q \rightarrow \mu$. The 3-valued semantics of PCTL associates a truth value in $\mathbb{B}_3$ for each formula φ in a state q of the MDP; we denote this by $[\![q, \varphi]\!]_{\mathcal{M}}$. We skip the formal semantics in the interests of space and the reader is referred to [8] [2].

Theorem 4 (Fecher-Leucker-Wolf [8]). *Given an MDP $\mathcal{M}$, and a PCTL formula φ, the value of $[\![q, \varphi]\!]_{\mathcal{M}}$ for each state q, can be computed in time polynomial in $|\mathcal{M}|$ and linear in $|\varphi|$, where $|\mathcal{M}|$ and $|\varphi|$ denote the sizes of $\mathcal{M}$ and φ, respectively.*

Simulation for MDPs, originally presented in [12] and adapted to the 3-valued semantics in [8], is defined as follows. A preorder $\sqsubseteq \subseteq Q \times Q$ is said to be a *simulation* iff whenever $q_1 \sqsubseteq q_2$ the following conditions hold.

- If $L(q_2, a) = \top$ or $\bot$ then $L(q_1, a) = L(q_2, a)$ for every proposition $a \in \mathrm{AP}$,
- If $q_1 \rightarrow \mu_1$ then there exists μ_2 such that $q_2 \rightarrow \mu_2$ and $\mu_1 \leq \mu_2$, where μ_1 and μ_2 are viewed as probability measures over the ordered measurable space $(Q, \mathcal{P}(Q), \sqsubseteq)$.

We say $q_1 \preceq q_2$ iff there is a simulation $\sqsubseteq$ such that $q_1 \sqsubseteq q_2$. A state q_1 in an MDP $(Q_1, \rightarrow_1, L_1)$ is simulated by a state q_2 in MDP $(Q_2, \rightarrow_2, L_2)$ iff there is a simulation $\sqsubseteq$ on the direct sum of the two MDP's (defined in the natural way) such that $(q_1, 0) \sqsubseteq (q_2, 1)$.

Remark 2. The ordering on probability measures used in simulation definition presented in [12,8] is defined using *weight functions*. However, the definition presented here is equivalent, as has been also observed in [7,20].

Finally, there is a close correspondence between simulation and the satisfaction of PCTL formulas according to the 3-valued interpretation.

Theorem 5 (Fecher-Leucker-Wolf [8]). *Consider q, q' states of MDP $\mathcal{M}$ such that $q \preceq q'$. For any formula φ, if $[\![q', \varphi]\!]_{\mathcal{M}} \neq ?$ then $[\![q, \varphi]\!]_{\mathcal{M}} = [\![q', \varphi]\!]_{\mathcal{M}}$* [3].

3.2 Abstraction by DTMCs

Abstraction, followed by progressive refinement, is one way to construct a small model that either proves the correctness of the system or demonstrates its failure to do so. Typically, the abstract model is defined with the help of an equivalence relation on the states of the system. Informally, the construction proceeds as

[2] In [8] PCTL semantics for MDPs is not given; however, this is very similar to the semantics for AMCs which is given explicitly.

[3] This theorem is presented only for AMC. But its generalization to MDPs can be obtained from the main observations given in [8]. See [4].

follows. For an MDP/DTMC $\mathcal{M} = (Q, \to, L)$ and equivalence relation $\equiv$ on Q, the abstraction is the MDP $\mathcal{A} = (Q_{\mathcal{A}}, \to_{\mathcal{A}}, L_{\mathcal{A}})$, where $Q_{\mathcal{A}} = \{[q]_{\equiv} \,|\, q \in Q\}$ is the set of equivalence classes of Q under $\equiv$, and $[q]_{\equiv}$ has a transition corresponding to each $q' \to \mu$ for $q' \in [q]_{\equiv}$.

However, as argued by Fecher-Leucker-Wolf [8], model checking $\mathcal{A}$ directly may not be feasible because it has large number of transitions and therefore a large size. It maybe beneficial to construct a further abstraction of $\mathcal{A}$ and analyze the further abstraction. In what follows, we have an MDP, which maybe obtained as outlined above, that we would like to (further) abstract; for the rest of this section let us fix this MDP to be $\mathcal{A} = (Q_{\mathcal{A}}, \to_{\mathcal{A}}, L_{\mathcal{A}})$. We will first present the Fecher-Leucker-Wolf proposal, then ours, and compare the approaches, discussing their relative merits.

Fecher et al. suggest that a set of transitions be approximated by *intervals* that bound the probability of transitioning from one state to the next, according to any of the non-deterministic choices present in $\mathcal{A}$. The resulting abstract model, which they call an *Abstract Markov Chain* (AMC) is formally defined as follows.

Definition 4. *The Abstract Markov Chain (AMC) associated with $\mathcal{A}$ is formally the tuple $\mathcal{M} = (Q_{\mathcal{M}}, \to_{\ell}, \to_{u}, L_{\mathcal{M}})$, where $Q_{\mathcal{M}} = Q_{\mathcal{A}}$ is the set of states, and $L_{\mathcal{M}} = L_{\mathcal{A}}$ is the labeling function on states. The lower bound transition $\to_{\ell}$ and upper bound transition $\to_{u}$ are both functions of type $Q_{\mathcal{M}} \to (Q_{\mathcal{M}} \to [0,1])$, and are defined as follows:*

$$q \to_{\ell} \mu \text{ iff } \forall q' \in Q_{\mathcal{M}}.\ \mu(q') = \min_{q \to_{\mathcal{A}} \nu} \nu(\{q'\})$$
$$q \to_{u} \mu \text{ iff } \forall q' \in Q_{\mathcal{M}}.\ \mu(q') = \max_{q \to_{\mathcal{A}} \nu} \nu(\{q'\})$$

Semantically, the AMC $\mathcal{M}$ is interpreted as an MDP having from each state q any transition $q \to \nu$, where ν is a probability measure that respects the bounds defined by $\to_{\ell}$ and $\to_{u}$. More precisely, if $q \to_{\ell} \mu_{\ell}$ and $q \to_{u} \mu_{u}$ then $\mu_{\ell} \leq \nu \leq \mu_{u}$, where $\leq$ is to be interpreted as pointwise ordering on functions.

Fecher et al. demonstrate that the AMC $\mathcal{M}$ (defined above) does indeed simulate $\mathcal{A}$, and using Theorem 5 the model checking results of $\mathcal{M}$ can be reflected to $\mathcal{A}$. The main advantage of $\mathcal{M}$ over $\mathcal{A}$ is that $\mathcal{M}$ can be model checked in time that is a polynomial in $2^{|Q_{\mathcal{M}}|} = 2^{|Q_{\mathcal{A}}|}$; model checking $\mathcal{A}$ may take time more than polynomial in $2^{|Q_{\mathcal{A}}|}$, depending on the number of transitions out of each state q.

We suggest using the upper bound construction, presented in Section 2.3, to construct *purely probabilistic* abstract models that do not have any nondeterminism. Let $(X, \sqsubseteq)$ be a tree-like partial order. Recall that the set of minimal elements of X, denoted by minimal(X), is given by minimal$(X) = \{x \in X \,|\, \forall y \in X.\ y \sqsubseteq x \ \Rightarrow\ x = y\}$.

Definition 5. *Consider the MDP $\mathcal{A} = (Q_{\mathcal{A}}, \to_{\mathcal{A}}, L_{\mathcal{A}})$. Let $(Q, \sqsubseteq)$ be a tree-like partial order, such that* minimal$(Q) = Q_{\mathcal{A}}$. *Let $\mathcal{Q} = (Q, \mathcal{P}(Q), \sqsubseteq)$ be the ordered measurable space over Q. Define the DTMC $\mathcal{D} = (Q_{\mathcal{D}}, \to_{\mathcal{D}}, L_{\mathcal{D}})$, where*

- $Q_{\mathcal{D}} = Q$,
- *For $q \in Q_{\mathcal{D}}$, let $\Gamma_q = \{\mu \mid \exists q' \in Q_{\mathcal{A}}.\ q' \sqsubseteq q$ and $q' \to_{\mathcal{A}} \mu\}$. Now, $q \to_{\mathcal{D}} \nabla(\Gamma_q)$, and*

– *For $q \in Q_{\mathcal{D}}$ and $a \in \mathrm{AP}$, if for any $q_1, q_2 \in Q_{\mathcal{A}}$ with $q_1 \sqsubseteq q$ and $q_2 \sqsubseteq q$, we have $L(q_1, a) = L(q_2, a)$ then $L(q, a) = L(q_1, a)$. Otherwise $L(q, a) =?$*

Proposition 2. *The DTMC $\mathcal{D}$ simulates the MDP $\mathcal{A}$, where $\mathcal{A}$ and $\mathcal{D}$ are as given in Definition 5.*

Proof. Consider the relation $R_{\sqsubseteq}$ over the states of the disjoint union of $\mathcal{A}$ and $\mathcal{D}$, defined as $R_{\sqsubseteq} = \{((q,0),(q,0)) \mid q \in Q_{\mathcal{A}}\} \cup \{((q',1),(q'',1)) \mid q', q'' \in Q_{\mathcal{D}}, q' \sqsubseteq q''\} \cup \{((q,0),(q',1)) \mid q \in Q_{\mathcal{A}}, q' \in Q_{\mathcal{D}}, q \sqsubseteq q'\}$. From the definition of $\mathcal{D}$, definition of simulation and the fact that ∇ is the least upper bound operator, it can be shown that $R_{\sqsubseteq}$ is a simulation. □

The minimality of our upper bound construction actually allows to conclude that $\mathcal{D}$ is as good as any DTMC abstraction can be on a given state space. This is stated precisely in the next proposition.

Proposition 3. *Let $\mathcal{A} = (Q_{\mathcal{A}}, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ be an MDP and $(Q, \sqsubseteq)$ be a tree-like partial order, such that* $\mathrm{minimal}(Q) = Q_{\mathcal{A}}$. *Consider the DTMC $\mathcal{D} = (Q_{\mathcal{D}}, \rightarrow_{\mathcal{D}}, L_{\mathcal{D}})$, as given in Definition 5. If $\mathcal{D}' = (Q_{\mathcal{D}}, \rightarrow'_{\mathcal{D}}, L_{\mathcal{D}})$ is a DTMC such that the relation $R_{\sqsubseteq}$ defined in the proof of Proposition 2 is a simulation of $\mathcal{A}$ by $\mathcal{D}'$ then $\mathcal{D}'$ simulates $\mathcal{D}$ also.*

Comparison with Abstract Markov Chains. Observe that any tree-like partial order $(Q, \sqsubseteq)$ such that $\mathrm{minimal}(Q) = Q_{\mathcal{A}}$ is of size at most $O(|Q_{\mathcal{A}}|)$; thus, in the worst case the time to model check $\mathcal{D}$ is exponentially smaller than the time to model check $\mathcal{M}$. Further, we have freedom to pick the partial order $(Q, \sqsubseteq)$. The following proposition says that adding more elements to the partial order on the abstract states does indeed result in a refinement.

Proposition 4. *Let $\mathcal{A} = (Q_{\mathcal{A}}, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ be an MDP and $(Q_1, \sqsubseteq_1)$ and $(Q_2, \sqsubseteq_2)$ be tree-like partial orders such that $Q_1 \subseteq Q_2$, $\sqsubseteq_2 \cap (Q_1 \times Q_1) = \sqsubseteq_1$, and $Q_{\mathcal{A}} =$* $\mathrm{minimal}(Q_1) = \mathrm{minimal}(Q_2)$. *Let $\mathcal{D}_1$ be a DTMC over $(Q_1, \sqsubseteq_1)$ and $\mathcal{D}_2$ a DTMC over $(Q_2, \sqsubseteq_2)$ as in Definition 5. Then, $\mathcal{D}_1$ simulates $\mathcal{D}_2$.*

Thus, one could potentially identify the appropriate tree-like partial order to be used for the abstract DTMC through a process of abstraction-refinement.

Finally, we can demonstrate that even though the DTMC $\mathcal{D}$ is exponentially more succinct than the AMC $\mathcal{M}$, there are examples where model checking $\mathcal{D}$ can give a more precise answer than $\mathcal{M}$.

Example 4. Consider an MDP $\mathcal{A}$ shown in Figure 2 where state 1 has two transitions one shown as solid edges and the other as dashed edges; transitions out of other states are not shown since they will not play a role. Suppose the atomic proposition a is $\top$ in $\{1,2\}$ and $\bot$ in $\{3,4\}$, and the proposition b is $\top$ in $\{1,3\}$ and $\bot$ in $\{2,4\}$. The formula $\varphi = \mathcal{P}_{\leq \frac{3}{4}}(Xa)$ evaluates to $\top$ in state 1.

The AMC $\mathcal{M}$ as defined in Definition 4, is shown in Figure 3. Now, because the distribution ν, given by $\nu(\{1\}) = \frac{1}{2}$, $\nu(\{2\}) = \frac{1}{2}$, $\nu(\{3\}) = 0$, and $\nu(\{4\}) = 0$

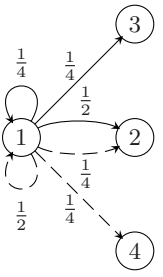


Fig. 2. Example MDP $\mathcal{A}$

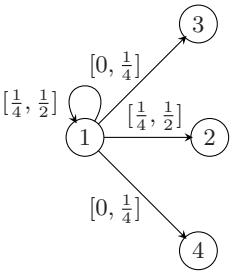


Fig. 3. AMC $\mathcal{M}$ corresponding to MDP $\mathcal{A}$

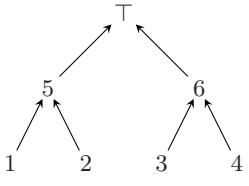


Fig. 4. Hasse diagram of partial order

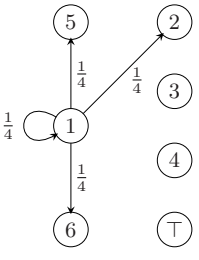


Fig. 5. Transition out of 1 in DTMC $\mathcal{D}$

satisfies the bound constraints out of 1 but violates the property φ, φ evaluates to ? in state 1 of $\mathcal{M}$.

Now consider the tree-like partial order shown in Figure 4; arrows in the figure point from the smaller element to the larger one. If we construct the DTMC $\mathcal{D}$ over this partial order as in Definition 5, the transition out of state 1 will be as shown in Figure 5. Observe also that proposition a is $\top$ in $\{1, 2, 5\}$, $\bot$ in $\{3, 4, 6\}$ and ? in state $\top$; and proposition b is $\top$ in $\{1, 3\}$, $\bot$ in $\{2, 4\}$ and ? in $\{5, 6, \top\}$. Now φ evaluates to $\top$ in state 1, because the measure of paths out of 1 that satisfy $X\neg a$ is $\frac{1}{4}$. Thus, by Theorem 5, $\mathcal{M}$ is not simulated by $\mathcal{D}$. It is useful to observe that the upper bound managed to capture the constraint that the probability of transitioning to either 3 or 4 from 1 is at least $\frac{1}{4}$. Constraints of this kind that relate to the probability of transitioning to a set of states, cannot be captured by the interval constraints of an AMC, but can be captured by upper bounds on appropriate partial orders.

4 Abstracting CTMCs

We now outline how our upper bound construction gives us a way to abstract CTMC by other CTMCs. We begin with recalling the definitions of CTMCs, simulation and logical preservation, before presenting our abstraction scheme.

4.1 Preliminaries

The formulas of CSL are built up over a finite set of atomic propositions AP and are inductively defined as follows.

$$\varphi ::= \text{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{P}_{\bowtie p}(\varphi\, \mathcal{U}^t\, \varphi)$$

where $a \in \text{AP}$, $\bowtie \in \{<, \leq, >, \geq\}$, $p \in [0,1]$, and t a positive real number.

The 3-valued semantics of CSL is defined over *Continuous Time Markov Chains* (CTMC), where in each state every atomic proposition gets a truth value in $\mathbb{B}_3$. Formally, let Q be a finite set of states and let $\mathcal{Q} = (Q, \mathcal{P}(Q))$ be a measure space. A (uniform) CTMC [4] $\mathcal{M}$ is a tuple $(Q, \rightarrow, L, E)$, where $\rightarrow : Q \rightarrow \mathcal{M}_{=1}(\mathcal{Q})$, $L : (Q \times \text{AP}) \rightarrow \mathbb{B}_3$ is a labeling function that assigns a value in $\mathbb{B}_3$ to each atomic proposition in each state, and $E \in \mathbb{R}_{\geq 0}$ is the *exit rate* from any state. We will say $q \rightarrow \mu$ to mean $(q, \mu) \in \rightarrow$. Due to lack of space the formal semantics of the CTMC is skipped; the reader is referred to [18].

CSL's 3-valued semantics associates a truth value in $\mathbb{B}_3$ for each formula φ in a state q of the CTMC; we denote this by $[\![q, \varphi]\!]_{\mathcal{M}}$. The formal semantics is skipped and can be found in [13]. The model checking algorithm presented in [1] for the 2-valued semantics, can be adapted to the 3-valued case.

Simulation for uniform CTMCs, originally presented in [3], has been adapted to the 3-valued setting in [13] and is defined in exactly the same way as simulation in a DTMC; since the exit rate is uniform, it does not play a role. Once again, we say q_1 is simulated by q_2, denoted as $q_1 \preceq q_2$, iff there is a simulation $\sqsubseteq$ such that $q_1 \sqsubseteq q_2$. Once again, there is a close correspondence between simulation and the satisfaction of CSL formulas according to the 3-valued interpretation.

Theorem 6 (Katoen-Klink-Leucker-Wolf [13]). *Consider any states q, q' of CTMC $\mathcal{M}$ such that $q \preceq q'$. For any formula φ, if $[\![q', \varphi]\!]_{\mathcal{M}} \neq ?$ then $[\![q, \varphi]\!]_{\mathcal{M}} = [\![q', \varphi]\!]_{\mathcal{M}}$.*

4.2 Abstracting Based on Upper Bounds

Abstraction can, once again, be accomplished by collapsing concrete states into a single abstract state on the basis of an equivalence relation on concrete states. The transition rates out of a single state can either be approximated by intervals giving upper and lower bounds, as suggested in [13], or by upper bound measures as we propose. Here we first present the proposal of Abstract CTMCs, where transition rates are approximated by intervals, before presenting our proposal. We conclude with a comparison of the two approaches.

Definition 6. *Consider a CTMC $\mathcal{M} = (Q_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, L_{\mathcal{M}}, E_{\mathcal{M}})$ with an equivalence relation $\equiv$ on $Q_{\mathcal{M}}$. An* Abstract CTMC *(ACTMC) [13] that abstracts $\mathcal{M}$ is a tuple $\mathcal{A} = (Q_{\mathcal{A}}, \rightarrow_{\ell}, \rightarrow_{u}, L_{\mathcal{A}}, E_{\mathcal{A}})$, where*

- *$Q_{\mathcal{A}} = \{[q] \mid q \in Q_{\mathcal{M}}\}$ is the set of equivalence classes of $\equiv$,*
- *$E_{\mathcal{A}} = E_{\mathcal{M}}$,*

[4] We only look at uniform CTMCs here; in general, any CTMC can be transformed in a uniform one that is weakly bisimilar to the original CTMC.

- *If for all* $q_1, q_2 \in [q]$, $L_{\mathcal{M}}(q_1, a) = L_{\mathcal{M}}(q_2, a)$ *then* $L_{\mathcal{A}}([q], a) = L_{\mathcal{M}}(q, a)$. *Otherwise,* $L_{\mathcal{A}}([q], a) = ?$,
- $\rightarrow_\ell : Q_{\mathcal{A}} \rightarrow (Q_{\mathcal{A}} \rightarrow [0,1])$ *where*

$$[q] \rightarrow_\ell \mu \text{ iff } \forall [q_1] \in Q_{\mathcal{A}}\ \mu([q_1]) = \min_{q' \in [q]\ \wedge\ q' \rightarrow_{\mathcal{A}} \nu} \nu([q_1])$$

- *Similarly,* $\rightarrow_u : Q_{\mathcal{A}} \rightarrow (Q_{\mathcal{A}} \rightarrow [0,1])$ *where*

$$[q] \rightarrow_u \mu \text{ iff } \forall [q_1] \in Q_{\mathcal{A}}\ \mu([q_1]) = \max_{q' \in [q]\ \wedge\ q' \rightarrow_{\mathcal{A}} \nu} \nu([q_1])$$

Semantically, at a state $[q]$*, the ACTMC can make a transition according to any transition rates that satisfy the lower and upper bounds defined by* $\rightarrow_\ell$ *and* $\rightarrow_u$*, respectively.*

Katoen et al. demonstrate that the ACTMC $\mathcal{A}$ (defined above) does indeed simulate $\mathcal{M}$, and using Theorem 6 the model checking results of $\mathcal{A}$ can be reflected to $\mathcal{M}$. The measure of paths reaching a set of states within a time bound t can be approximated using ideas from [2], and this can be used to answer model checking question for the ACTMC (actually, the path measures can only be calculated upto an error).

Like in Section 3.2, we will now show how the upper bound construction can be used to construct (standard) CTMC models that abstract the concrete system. Before presenting this construction, it is useful to define how to lift a measure on a set with an equivalence relation $\equiv$, to a measure on the equivalence classes of $\equiv$.

Definition 7. *Given a measure* μ *on* $(Q, \mathcal{P}(Q))$ *and equivalence* $\equiv$ *on* Q*, the lifting of* μ *(denoted by* $[\mu]$*) to the set of equivalence classes of* Q *is defined as* $[\mu](\{[q]\}) = \mu(\{q' \mid q' \equiv q\})$.

Definition 8. *Let* $\mathcal{M} = (Q_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, L_{\mathcal{M}}, E_{\mathcal{M}})$ *be a CTMC with an equivalence relation* $\equiv$ *on* $Q_{\mathcal{M}}$*. Let* $(Q, \sqsubseteq)$ *be a tree-like partial order with* $\top$*, such that* $\text{minimal}(Q) = \{[q] \mid q \in Q_{\mathcal{M}}\}$*. Let* $\mathcal{Q} = (Q, \mathcal{P}(Q), \sqsubseteq)$ *be the ordered measurable space over* Q*. Define the CTMC* $\mathcal{C} = (Q_{\mathcal{C}}, \rightarrow_{\mathcal{C}}, L_{\mathcal{C}}, E_{\mathcal{C}})$*, where*

- $Q_{\mathcal{C}} = Q$,
- $E_{\mathcal{C}} = E_{\mathcal{M}}$,
- *For* $q \in Q_{\mathcal{C}}$*, let* $\Gamma_q = \{[\mu] \mid \exists q' \in Q_{\mathcal{A}}.\ [q'] \sqsubseteq q \text{ and } q' \rightarrow_{\mathcal{A}} \mu\}$*. Now,* $q \rightarrow_{\mathcal{C}} \nabla(\Gamma_q)$*, and*
- *If for all* $q_1, q_2 \in Q_{\mathcal{M}}$ *such that* $[q_1] \sqsubseteq q$ *and* $[q_2] \sqsubseteq q$*,* $L_{\mathcal{M}}(q_1, a) = L_{\mathcal{M}}(q_2, a)$ *then* $L_{\mathcal{C}}(q, a) = L_{\mathcal{M}}(q_1, a)$*. Otherwise,* $L_{\mathcal{C}}(q, a) = ?$.

Once again, from the properties of least upper bounds, and definition of simulation, we can state and prove results analogous to Propositions 2 and 3. That is the CTMC $\mathcal{C}$ does indeed abstract $\mathcal{M}$ and it is the best possible on a given state space; the formal statements and proofs are skipped in the interests of space.

Comparison with Abstract CTMCs. All the points made when comparing the DTMC abstraction with the AMC abstraction scheme, also apply here. That is, the size of $\mathcal{C}$ is exponentially smaller than the size of the ACTMC $\mathcal{A}$. Moreover, we can choose the tree-like partial order used in the construction of $\mathcal{C}$ through a process of abstraction refinement. And finally, Example 4 can be modified to demonstrate that there are situations where the CTMC $\mathcal{C}$ gives a more precise result than the ACTMC $\mathcal{A}$. However, in the context of CTMCs there is one further advantage. ACTMCs can only be model checked approximately, while CTMCs can be model checked exactly. While it is not clear how significant this might be in practice, from a theoretical point of view, it is definitely appealing.

5 Conclusions

Our main technical contribution is the construction of least upper bounds for probability measures on measure spaces equipped with a partial order. We have developed an exact characterization of underlying orderings for which the induced ordering on probability measures admits the existence of least upper bounds, and provided a natural construction for defining them. We showed how these upper bound constructions can be used to abstract DTMCs, MDPs, and CTMCs by models that are purely probabilistic. In some situations, our abstract models yield more precise model checking results than previous proposals for abstraction. Finally, we believe that the absence of nondeterminism in the abstract models we construct might make their model-checking more feasible.

In terms of future work, it would be important to evaluate how these abstraction techniques perform in practice. In particular, the technique of identifying the right tree-like state space for the abstract models using abstraction-refinement needs to be examined further.

References

1. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model checking continuous-time Markov chains. ACM TOCL 1, 162–170 (2000)
2. Baier, C., Haverkrot, B., Hermanns, H., Katoen, J.-P.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 61–76. Springer, Heidelberg (2004)
3. Baier, C., Hermanns, H., Katoen, J.-P., Wolf, V.: Comparative branching-time semantics for Markov chains. Inf. and Comp. 200, 149–214 (2005)
4. Chadha, R., Viswanathan, M., Viswanathan, R.: Least upper bounds for probability measures and their applications to abstractions. Technical Report UIUCDCS-R-2008-2973, UIUC (2008)
5. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001)
6. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reduction and refinement strategies for probabilistic analysis. In: Hermanns, H., Segala, R. (eds.) PROBMIV 2002. LNCS, vol. 2399, pp. 57–76. Springer, Heidelberg (2002)

7. Desharnais, J.: Labelled Markov Processes. PhD thesis, McGill University (1999)
8. Fecher, H., Leucker, M., Wolf, V.: Don't know in probabilistic systems. In: Proc. of SPIN, pp. 71–88 (2006)
9. Huth, M.: An abstraction framework for mixed non-deterministic and probabilistic systems. In: Validation of Stochastic Systems: A Guide to Current Research, pp. 419–444 (2004)
10. Huth, M.: On finite-state approximants for probabilistic computation tree logic. TCS 346, 113–134 (2005)
11. Jones, C.: Probabilistic Non-determinism. PhD thesis, University of Edinburgh (1990)
12. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: Proc. of LICS, pp. 266–277 (1991)
13. Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for continuous-time Markov chains. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 311–324. Springer, Heidelberg (2007)
14. Kwiatkowska, M., Norman, G., Parker, D.: Game-based abstraction for Markov decision processes. In: Proc. of QEST, pp. 157–166 (2006)
15. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer, Heidelberg (2004)
16. Monniaux, D.: Abstract interpretation of programs as Markov decision processes. Science of Computer Programming 58, 179–205 (2005)
17. Norman, G.: Analyzing randomized distributed algorithms. In: Validation of Stochastic Systems: A Guide to Current Research, pp. 384–418 (2004)
18. Rutten, J.M., Kwiatkowska, M., Norman, G., Parker, D.: Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems. AMS (2004)
19. Saheb-Djahromi, N.: Probabilistic LCF. In: Winkowski, J. (ed.) MFCS 1978. LNCS, vol. 64, pp. 442–451. Springer, Heidelberg (1978)
20. Segala, R.: Probability and nondeterminism in operational models of concurrency. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 64–78. Springer, Heidelberg (2006)
21. Sen, K., Viswanathan, M., Agha, G.: Model checking Markov chains in the presence of uncertainties. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 394–410. Springer, Heidelberg (2006)
22. Wachter, B., Zhang, L., Hermanns, H.: Probabilistic model checking modulo theories. In: Proc. of QEST (2007)
23. Younes, H., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking: An empirical study. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 46–60. Springer, Heidelberg (2004)
24. Younes, H., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)

Abstraction for Stochastic Systems by Erlang's Method of Stages*

Joost-Pieter Katoen[1], Daniel Klink[1], Martin Leucker[2], and Verena Wolf[3]

[1] RWTH Aachen University
[2] TU Munich
[3] EPF Lausanne

Abstract. This paper proposes a novel abstraction technique based on Erlang's method of stages for continuous-time Markov chains (CTMCs). As abstract models *Erlang-k interval processes* are proposed where state residence times are governed by Poisson processes and transition probabilities are specified by intervals. We provide a three-valued semantics of CSL (Continuous Stochastic Logic) for Erlang-k interval processes, and show that both affirmative and negative verification results are preserved by our abstraction. The feasibility of our technique is demonstrated by a quantitative analysis of an enzyme-catalyzed substrate conversion, a well-known case study from biochemistry.

1 Introduction

This paper is concerned with a novel abstraction technique for timed probabilistic systems, in particular continuous-time Markov chains, CTMCs for short. These models are omnipresent in performance and dependability analysis, as well as in areas such as systems biology. In recent years, they have been the subject of study in concurrency theory and model checking. CTMCs are a prominent operational model for stochastic process algebras [13] and have a rich theory of behavioral (both linear-time and branching-time) equivalences, see, e.g., [4,26]. Efficient numerical, as well as simulative verification algorithms have been developed [1,3,27] and have become an integral part of dedicated probabilistic model checkers such as PRISM and act as backend to widely accepted performance analysis tools like GreatSPN and the PEPA Workbench.

Put in a nutshell, CTMCs are transition systems whose transitions are equipped with discrete probabilities and state residence times are determined by negative exponential distributions. Like transition systems, they suffer from the state-space explosion problem. To overcome this problem, several abstraction-based approaches have recently been proposed. Symmetry reduction [20], bisimulation minimization [16], and advances in quotienting algorithms for simulation pre-orders [28] show encouraging experimental results. Tailored abstraction techniques for regular infinite-state CTMCs have been reported [22], as well as bounding techniques that approximate CTMCs by ones having a special structure allowing closed-form solutions [21]. Predicate abstraction techniques have been extended to (among others) CTMCs [14]. There is a wide range of

* The research has been partially funded by the DFG Research Training Group 1298 (AlgoSyn), the Swiss National Science Foundation under grant 205321-111840 and the EU FP7 project Quasimodo.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 279–294, 2008.

related work on abstraction of discrete-time probabilistic models such as MDPs, see e.g., [9,8,19]. Due to the special treatment of state residence times, these techniques are not readily applicable to the continuous-time setting.

This paper generalizes and improves upon our three-valued abstraction technique for CTMCs [17]. We adopt a three-valued semantics, i.e., an interpretation in which a logical formula evaluates to either true, false, or indefinite. In this setting, abstraction preserves a simulation relation on CTMCs and is conservative for both positive and negative verification results. If the verification of the abstract model yields an indefinite answer, the validity in the concrete model is unknown. In order to avoid the grouping of states with distinct residence time distributions, the CTMC is *uniformized* prior to abstraction. This yields a weak bisimilar CTMC [4] in which all states have identical residence time distributions. Transition probabilities of single transitions are abstracted by intervals, yielding continuous-time variants of interval DTMCs [10,24].

This, however, may yield rather coarse abstractions (see below). This paper suggests to overcome this inaccuracy. The crux of our approach is to collapse *transition sequences* of a given fixed length k, say. Our technique in [17] is obtained if $k{=}1$. This paper presents the theory of this abstraction technique, shows its correctness, and shows its application by a quantitative analysis of an enzyme-catalyzed substrate conversion, a well-known case study from biochemistry [5].

Let us illustrate the main idea of the abstraction by means of an example. Consider the CTMC shown on the right (top). Intuitively, a CTMC can be considered as a transition system whose transitions are labeled with *transition probabilities*. Moreover, a CTMC comes with an *exit rate* identifying the *residence times* of the states (one, say), which is exponentially distributed. The essence of CTMC model checking is to compute the probability to reach a certain set of goal states within a given deadline [3].

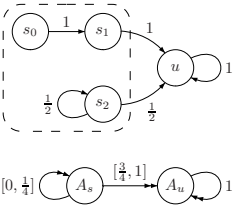


Fig. 1.

A rather common approach to abstraction is to partition the state space into classes, e.g., let us group states s_0, s_1, and s_2 into the abstract state A_s, and u into A_u. The probability to move from A_s to A_u by a single transition is either 0, $\frac{1}{2}$, or 1, as the respective (time-abstract) probability to move to u *in one transition* is 0, 1, and $\frac{1}{2}$. The approach in [17] yields the interval $[0,1]$ for the transition from A_s to A_u. This is not very specific. A more narrow interval is obtained when considering two consecutive transitions. Then, the probability from A_s to A_u is 1 or $\frac{3}{4}$. Using intervals, this yields the two-state abstract structure depicted above (bottom).

Put in a nutshell, the abstraction technique proposed in this paper is to generalize this approach towards considering transition sequences of a given length $k > 0$, say. State residence times are, however, then no longer exponentially distributed, but Erlang-k distributed. Moreover, taking each time k steps at once complicates the exact calculation of time-bounded reachability probabilities: Let us consider first the case that n is the number of transitions taken in the concrete system to reach a certain goal state. Let ℓ and j be such that $n = \ell{\cdot}k{+}j$ and $j \in \{0,\ldots,k{-}1\}$. Clearly, the number of transitions in the abstract system corresponds exactly to a multiple of the number of transitions in the concrete system, only if the remainder j equals 0. As this is generally not the case,

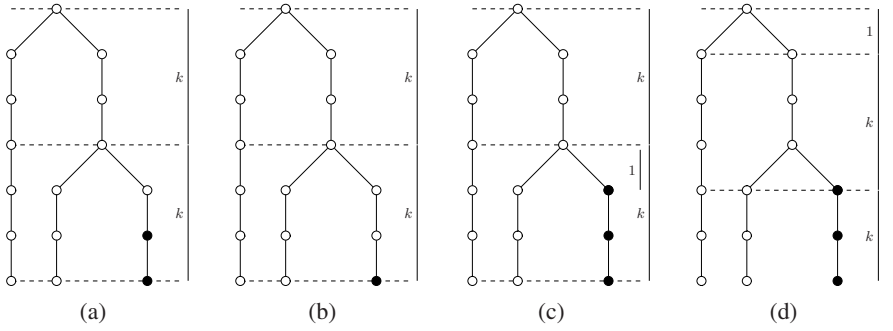


Fig. 2. Reaching goals in stages of length k

we restrict to computing lower and upper bounds for the probability of reaching a set of goal states. Let us be more precise: Consider the tree of state sequences as shown in Fig. 2(a). Let the black nodes denote the set of goal states. Taking the right branch, 5 transitions are needed to reach a goal state. For $k = 3$, this implies that 2 abstract transitions lead to a goal state. However, as $2 \cdot 3 = 6$, computing with 2 transitions and Erlang-3 distributed residence times will not give the exact probability for reaching a goal state, but, as we show, a *lower bound*. Intuitively, the probability for reaching a goal state in Fig. 2(b) is computed. For an upper bound, one might first consider all states from the fourth state on in the right branch as goal states. This would give a rather coarse upper bound. We follow instead the idea depicted in Fig. 2(c): We consider 2 transitions for reaching the goal state, however, use the Erlang-3 distribution for assessing the first transition, but use the Erlang-1 distribution for assessing the last transition of a sequence of transitions. That is, we compute the reachability probability for the goal states as depicted in Fig. 2(c). Technically, it is beneficial to understand the situation as depicted in Fig. 2(d), i.e., to first consider one transition with Erlang-1 distribution and then to consider a sequence of transitions which are Erlang-k distributed.

Outline of the paper. Section 2 gives some necessary background. We introduce Erlang-k interval processes in Section 3 which serve as abstract model for CTMCs in Section 4. In Section 5, we focus on reachability analysis of Erlang-k interval processes and utilize it for model checking in Section 6. The feasibility of our approach is demonstrated in Section 7 by a case study from biology and Section 8 concludes the paper. A full version with detailed proofs can be found in [18].

2 Preliminaries

Let X be a finite set. For $Y, Y' \subseteq X$ and function $f : X \times X \to \mathbb{R}$ let $f(Y, Y') := \sum_{y \in Y, y' \in Y'} f(y, y')$ (for singleton sets, brackets may be omitted). The function $f(x, \cdot)$ is given by $x' \mapsto f(x, x')$ for all $x \in X$. Function f is a *distribution on* X iff $f : X \to [0, 1]$ and $f(X) := \sum_{x \in X} f(x) = 1$. The set of all distributions on X is denoted by $distr(X)$. Let AP be a fixed, finite set of atomic propositions and $\mathbb{B}_2 := \{\bot, \top\}$ the two-valued truth domain.

Continuous-time Markov chains. A (uniform) *CTMC* $\mathcal{C}$ is a tuple $(S, \mathbf{P}, \lambda, L, s_0)$ with a finite non-empty set of states S, a transition probability function $\mathbf{P} : S \times S \to [0,1]$ such that $\mathbf{P}(s, S) = 1$ for all $s \in S$, an exit rate $\lambda \in \mathbb{R}_{>0}$, a labeling function $L : S \times AP \to \mathbb{B}_2$, and an initial state $s_0 \in S$. This definition deviates from the literature as i) we assume a uniform exit rate and ii) we separate the discrete-time behavior specified by $\mathbf{P}$ and the residence times determined by λ. Restriction i) is harmless, as every (non-uniform) CTMC can be transformed into a weak bisimilar, uniform CTMC by adding self-loops [25]. For ii), note that $\mathbf{P}(s, s')(1-e^{\lambda t})$ equals the probability to reach s' from s in one step and within time interval $[0, t)$. Thus, the above formulation is equivalent to the standard one. The expected state residence time is $1/\lambda$. Let $\mathbf{P}^k(s, s')$ denote the time-abstract probability to enter state s' after k steps while starting from s, which is obtained by taking the kth-power of $\mathbf{P}$ (understood as a transition probability matrix).

We recall some standard definitions for Markov chains [11,23]. An infinite *path* σ is a sequence $s_0\, t_0\, s_1\, t_1\, \ldots$ with $s_i \in S$, $\mathbf{P}(s_i, s_{i+1}) > 0$ and $t_i \in \mathbb{R}_{>0}$ for $i \in \mathbb{N}$. The time stamps t_i denote the residence time in state s_i. Let $\sigma@t$ denote the state of a path σ occupied at time t, i.e. $\sigma@t = s_i$ with i the smallest index such that $t < \sum_{j=0}^{i} t_j$. The set of all (infinite) paths in $\mathcal{C}$ is denoted by $Path_{\mathcal{C}}$. Let Pr be the probability measure on sets of paths that results from the standard cylinder set construction.

Poisson processes. Let $(N_t)_{t\geq 0}$ be a counting process and let the corresponding interarrival times be independent and identically exponentially distributed with parameter $\lambda > 0$. Then $(N_t)_{t\geq 0}$ is a *Poisson process* and the number k of arrivals in time interval $[0, t)$ is Poisson distributed, i.e., $P(N_t = k) = e^{-\lambda t}(\lambda t)^k/k!$. The time until k arrivals have occurred is Erlang-k distributed, i.e., $F_{\lambda,k}(t) := P(T_k \leq t) = 1 - \sum_{i=0}^{k-1} e^{-\lambda t}\frac{(\lambda t)^i}{i!}$ where T_k is the time instant of the k-th arrival in $(N_t)_{t\geq 0}$. Consequently, the probability that $(N_t)_{t\geq 0}$ is in the range $\{k, k+1, \ldots, k+\ell-1\}$, $\ell \geq 1$ is given by

$$\psi_{\lambda,t}(k, \ell) := P(T_k \leq t < T_{k+\ell}) = \sum_{i=k}^{k+\ell-1} e^{-\lambda t}\frac{(\lambda t)^i}{i!} \ .$$

A CTMC $\mathcal{C} = (S, \mathbf{P}, \lambda, L, s_0)$ can be represented as a discrete-time Markov chain with transition probabilities $\mathbf{P}$ where the times are implicitly driven by a Poisson process with parameter λ, i.e., the probability to reach state s' from s within $[0, t)$ is:

$$\sum_{i=0}^{\infty} \mathbf{P}^i(s, s') \cdot e^{-\lambda t}\frac{(\lambda t)^i}{i!} \ .$$

This relationship can be used for an efficient transient analysis of CTMCs and is known as *uniformization*. A truncation point of the infinite sum can be calculated such that the approximation error is less than an a priori defined error bound [25].

Continuous Stochastic Logic. CSL [1,3] extends PCTL [12] by equipping the until-operator with a time bound. Its syntax is given by:

$$\varphi ::= \mathit{true} \mid a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathcal{P}_{\bowtie p}(\varphi\, \mathcal{U}^I \varphi)$$

where $I \in \{[0,t), [0,t], [0,\infty) \mid t \in \mathbb{R}_{>0}\}$, $\bowtie \in \{<, \leq, \geq, >\}$, $p \in [0,1]$ and $a \in AP$. The formal semantics of CSL is given in Table 1. CSL model checking [3] is performed inductively on the structure of φ like for CTL model checking. Checking time-bounded until-formulas boils down to computing time-bounded reachability probabilities. These

Table 1. Semantics of CSL

$[\![true]\!](s) = \top$	$[\![a]\!](s) = L(s,a)$
$[\![\varphi_1 \wedge \varphi_2]\!](s) = [\![\varphi_1]\!](s) \sqcap [\![\varphi_2]\!](s)$	$[\![\neg\varphi]\!](s) = ([\![\varphi]\!](s))^c$
$[\![\mathcal{P}_{\bowtie p}(\varphi_1 \mathcal{U}^I \varphi_2)]\!](s) = \top$, iff $Pr(\{\sigma \in Path_s^{\mathcal{M}} \mid [\![\varphi_1 \mathcal{U}^I \varphi_2]\!](\sigma) = \top\}) \bowtie p$	
$[\![\varphi_1 \mathcal{U}^I \varphi_2]\!](\sigma) = \top$, iff $\exists\, t \in I : ([\![\varphi_2]\!](\sigma@t) = \top \wedge\ \forall\, t' \in [0,t) : [\![\varphi_1]\!](\sigma@t') = \top)$	

probabilities can be obtained by a reduction to transient analysis yielding a time complexity in $\mathcal{O}(|S|^2 \lambda t)$ where t is the time bound.

Three-valued domain. Let $\mathbb{B}_3 := \{\bot, ?, \top\}$ be the complete lattice with ordering $\bot < ? < \top$, meet ($\sqcap$) and join ($\sqcup$) as expected, and complementation $\cdot^c$ such that $\top$ and $\bot$ are complementary to each other and $?^c = ?$. When a formula evaluates to $\bot$ or $\top$, the result is *definitely* true or false respectively, otherwise it is *indefinite*.

3 Erlang-k Interval Processes

Erlang-k interval processes are determined by two ingredients: a discrete probabilistic process with intervals of transition probabilities (like in [10,24]) and a Poisson process. The former process determines the probabilistic branching whereas residence times are governed by the latter. More precisely, the state residence time is the time until j further arrivals occur according to the Poisson process where $j \in \{1, \ldots, k\}$ is nondeterministically chosen. Thus, the residence times are Erlang-j distributed.

Definition 1 (Erlang-k interval process). *An* Erlang-k interval process *is a tuple* $\mathcal{E} = (S, \mathbf{P}_l, \mathbf{P}_u, \lambda, k, L, s_0)$*, with* S *and* $s_0 \in S$ *as before, and* $\mathbf{P}_l, \mathbf{P}_u : S \times S \to [0,1]$*, transition probability bounds such that for all* $s \in S$*:* $\mathbf{P}_l(s,S) \leq 1 \leq \mathbf{P}_u(s,S)$*,* $\lambda \in \mathbb{R}_{>0}$*, a parameter of the associated Poisson process,* $k \in \mathbb{N}^+$*, and* $L : S \times AP \to \mathbb{B}_3$*.*

An Erlang-1 interval process is an *abstract continuous-time Markov chain* (ACTMC) [17]. If additionally all intervals are singletons, the process is equivalent to a CTMC with $\mathbf{P}_l = \mathbf{P}_u = \mathbf{P}$. The set of transition probability functions for $\mathcal{E}$ is:

$$\mathbf{T}_{\mathcal{E}} := \{\mathbf{P} : S \times S \to [0,1] \mid \forall s \in S :\ \mathbf{P}(s,S) = 1, \\ \forall s, s' \in S :\ \mathbf{P}_l(s,s') \leq \mathbf{P}(s,s') \leq \mathbf{P}_u(s,s')\}$$

Let $\mathbf{T}_{\mathcal{E}}(s) := \{\mathbf{P}(s,\cdot) \mid \mathbf{P} \in \mathbf{T}_{\mathcal{E}}\}$ be the set of distributions in s.

Paths in Erlang-k interval processes. A *path* σ in $\mathcal{E}$ is an infinite sequence $s_0 t_0 s_1 t_1 \ldots$ with $s_i \in S, t_i \in \mathbb{R}_{>0}$ for which there exists $\mathbf{P}_0, \mathbf{P}_1, \ldots \in \mathbf{T}_{\mathcal{E}}$ such that $\mathbf{P}_i(s_i, s_{i+1}) > 0$ for all $i \in \mathbb{N}$. A *path fragment* ξ is a prefix of a path that ends in a state denoted $\xi{\downarrow}$. The set of all path fragments ξ (untimed path fragments) in $\mathcal{E}$ is denoted by $Pathf_{\mathcal{E}}$ ($uPathf_{\mathcal{E}}$, respectively) whereas the set of paths is denoted by $Path_{\mathcal{E}}$.

We depict Erlang-k interval processes by drawing the state-transition graph of the discrete part, i.e., the associated interval DTMC with transitions labeled by $[\mathbf{P}_l(s,s'),$

$\mathbf{P}_u(s,s')]$ (see, e.g., Fig. 3). The Poisson process that determines the residence times, as well as the marking of the initial state are omitted.

Normalization. Erlang-k interval process $\mathcal{E}$ is called *delimited*, if every possible selection of a transition probability in a state can be extended to a distribution [17], i.e., if for any $s, s' \in S$ and $p \in [\mathbf{P}_l(s,s'), \mathbf{P}_u(s,s')]$, we have $\mu(s') = p$ for some $\mu \in \mathbf{T}_{\mathcal{E}}(s)$. An Erlang-$k$ interval process $\mathcal{E}$ can be normalized into a delimited one $norm(\mathcal{E})$ such that $\mathbf{T}_{norm(\mathcal{E})} = \mathbf{T}_{\mathcal{E}}$. Formally, $norm(\mathcal{E}) = (S, \tilde{\mathbf{P}}_l, \tilde{\mathbf{P}}_u, \lambda, k, L, s_0)$ with for all $s, s' \in S$:

$$\begin{aligned} \tilde{\mathbf{P}}_l(s,s') &= \max\{\mathbf{P}_l(s,s'), 1 - \mathbf{P}_u(s, S \setminus \{s'\})\} \quad \text{and} \\ \tilde{\mathbf{P}}_u(s,s') &= \min\{\mathbf{P}_u(s,s'), 1 - \mathbf{P}_l(s, S \setminus \{s'\})\}. \end{aligned}$$

Example 1. The Erlang-k interval process in Fig. 3, left, is delimited. Selecting $\frac{1}{4}$ for the transition from s to u_2 yields a non-delimited process (Fig. 3, middle). Applying normalization results in the Erlang-k interval process shown in Fig. 3, right.

An Erlang-k interval process contains two sources of nondeterminism: in each state, (i) a distribution according to the transition probability intervals, and (ii) the number $j \in \{1, \ldots, k\}$ of arrivals in the Poisson process may be chosen. As usual, nondeterminism is resolved by a scheduler:

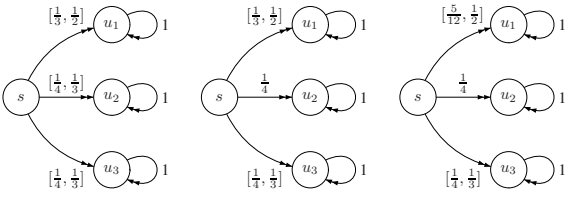


Fig. 3. Normalization

Definition 2 (Scheduler). *Let $\mathcal{E}$ be an Erlang-k interval process. A* history-dependent deterministic scheduler *is a function $D : uPathf_{\mathcal{E}} \to distr(S) \times \{1, \ldots, k\}$ such that $D(\xi) \in \mathbf{T}_{\mathcal{E}}(\xi\downarrow) \times \{1, \ldots, k\}$ for all $\xi \in uPathf_{\mathcal{E}}$. The set of all history-dependent deterministic schedulers of $\mathcal{E}$ is denoted as $\mathcal{HD}^{\mathcal{E}}$.*

Note that a richer class of schedulers is obtained if the scheduler's choice may also depend on the residence times of the states visited so far. We show below that the class of history-dependent deterministic schedulers suffices when Erlang-k interval processes are used for abstracting CTMCs.

Probability measure. For Erlang-k interval process $\mathcal{E}$, let $\Omega = Path_{\mathcal{E}}$ be the sample space and $\mathcal{B}$ the Borel field generated by the basic cylinder sets $\mathcal{C}(s_0\, I_0 \ldots I_{n-1}\, s_n)$ where $s_i \in S$, $0 \le i \le n$ and $I_\ell = [0, x_\ell) \subseteq \mathbb{R}_{\ge 0}$ is a non-empty interval for $0 \le \ell < n$. The set $\mathcal{C}(s_0\, I_0 \ldots I_{n-1}\, s_n)$ contains all paths of $\mathcal{E}$ with prefix $\hat{s}_0\, t_0 \ldots t_{n-1}\, \hat{s}_n$ such that $s_i = \hat{s}_i$ and $t_\ell \in I_\ell$. A scheduler $D \in \mathcal{HD}^{\mathcal{E}}$ induces a probability space $(\Omega, \mathcal{B}, Pr^D)$ where Pr^D is uniquely given by $Pr^D(C(s_0)) := 1$ and for $n \ge 0$

$$\begin{aligned} Pr^D(C(s_0\, I_0 \ldots I_n\, s_{n+1})) &:= Pr^D(C(s_0\, I_0 \ldots I_{n-1}\, s_n)) \cdot F_{\lambda, j_n}(\sup I_n) \cdot \mu_n(s_{n+1}) \\ &= \textstyle\prod_{i=0}^{n} \left(F_{\lambda, j_i}(\sup I_i) \cdot \mu_i(s_{i+1})\right) \end{aligned}$$

where $(\mu_i, j_i) =: D(s_0\, s_1\, \ldots\, s_i)$. Additionally, we define the time-abstract probability measure induced by D as $Pr^D_{ta}(C(s_0)) := 1$ and

$$Pr^D_{ta}(C(s_0\, I_0\, \ldots\, I_n\, s_{n+1})) := \prod_{i=0}^{n} \mu_i(s_{i+1}).$$

We are interested in the supremum/infimum (ranging over all schedulers) of the probability of measurable sets of paths. Clearly, the choice of j_i, the number of steps in the associated Poisson process in state s_i, may influence such quantities. For instance, on increasing j_i, time-bounded reachability probabilities will decrease as the expected state residence time (in s_i) becomes longer. We discuss the nondeterministic choice in the Poisson process in subsequent sections, and now focus on the choice of distribution μ_i according to the probability intervals.

Definition 3 (Extreme distributions). *Let $\mathcal{E}$ be an Erlang-k interval process, $s \in S$ and $S' \subseteq S$. We define extr$(\mathbf{P}_l, \mathbf{P}_u, S', s) \subseteq \mathbf{T}_{\mathcal{E}}(s)$ such that $\mu \in extr(\mathbf{P}_l, \mathbf{P}_u, S', s)$ iff either $S' = \emptyset$ and $\mu = \mathbf{P}_l(s, \cdot) = \mathbf{P}_u(s, \cdot)$ or one of the following conditions holds*[1]*:*

- $\exists s' \in S' : \mu(s') = \mathbf{P}_l(s, s')$ *and* $\mu \in$ *extr*$(\mathbf{P}_l, \mathbf{P}_u[(s, s') \mapsto \mu(s')], S' \setminus \{s'\}, s)$
- $\exists s' \in S' : \mu(s') = \mathbf{P}_u(s, s')$ *and* $\mu \in$ *extr*$(\mathbf{P}_l[(s, s') \mapsto \mu(s')], \mathbf{P}_u, S' \setminus \{s'\}, s)$

We call $\mu \in \mathbf{T}_{\mathcal{E}}(s)$ an extreme distribution *if $\mu \in$ extr$(\mathbf{P}_l, \mathbf{P}_u, S, s)$.*

A scheduler $D \in \mathcal{HD}^{\mathcal{E}}$ is called *extreme* if all choices $D(\xi)$ are extreme distributions. For a subset $\mathcal{D} \subseteq \mathcal{HD}^{\mathcal{E}}$ let $\mathcal{D}_{extr} \subseteq \mathcal{D}$ denote the subset of all extreme schedulers in $\mathcal{D}$.

Theorem 1 (Extrema). *Let $\mathcal{E}$ be an Erlang-k interval process and $\mathcal{D} \subseteq \mathcal{HD}^{\mathcal{E}}$. For every measurable set $Q \in \mathcal{B}$ of the induced probability space:*

$$\inf_{D \in \mathcal{D}_{extr}} Pr^D(Q) = \inf_{D \in \mathcal{D}} Pr^D(Q), \quad \sup_{D \in \mathcal{D}_{extr}} Pr^D(Q) = \sup_{D \in \mathcal{D}} Pr^D(Q).$$

4 Abstraction

This section makes the abstraction by stages as motivated in the introduction precise. We define an abstraction operator based on the idea of partitioning the concrete states to form abstract states. This yields an Erlang-k interval process. Moreover, we introduce a simulation relation relating one transition in the abstract system to a sequence of k transitions in the concrete system. We show that the abstraction operator yields an Erlang-k interval process simulating the original CTMC.

Definition 4 (Abstraction). *Let abstr$(\mathcal{C}, \mathcal{A}, k) := (\mathcal{A}, \mathbf{P}_l, \mathbf{P}_u, \lambda, k, L', A_0)$ be the* abstraction *of CTMC $\mathcal{C} = (S, \mathbf{P}, \lambda, L, s_0)$ induced by partitioning $\mathcal{A} = \{A_0, \ldots, A_n\}$ of S and $k \in \mathbb{N}^+$ such that for all $1 \le i, j \le n$:*

- $\mathbf{P}_l(A_i, A_j) = \min_{s \in A_i} \mathbf{P}^k(s, A_j)$, *and* $\mathbf{P}_u(A_i, A_j) = \max_{s \in A_i} \mathbf{P}^k(s, A_j)$
- $L'(A, a) = \begin{cases} \top & \text{if for all } s \in A : L(s, a) = \top \\ \bot & \text{if for all } s \in A : L(s, a) = \bot \\ ? & \text{otherwise} \end{cases}$
- $A_0 \in \mathcal{A}$ *with* $s_0 \in A_0$.

[1] $f[y \mapsto x]$ denotes the function that agrees everywhere with f except at y where it equals x.

Lemma 1. *For any CTMC $\mathcal{C}$, any partitioning $\mathcal{A}$ of S and $k \in \mathbb{N}^+$, abstr$(\mathcal{C}, \mathcal{A}, k)$ is an Erlang-k interval process.*

Example 2. Reconsider the CTMC $\mathcal{C}$ from Section 1 (Fig. 1), top, with exit rate $\lambda = 1$ and partitioning $\{A_s, A_u\}$ with $A_s = \{s_0, s_1, s_2\}$, $A_u = \{u\}$. As remarked above, in the Erlang-1 interval process *abstr*$(\mathcal{C}, \{A_s, A_u\}, 1)$ (not shown) the probability interval for a transition from A_s to A_u is $[0, 1]$. However, choosing $k = 2$ yields smaller intervals. The resulting Erlang-2 interval process is depicted in Fig. 1, bottom. The plot in Fig. 4 shows the probability to reach $A_u = \{u\}$ within t time units if the Erlang-2 interval process starts at time 0 in A_s and the CTMC in s_0, s_1 or s_2, respectively. For the Erlang-2 interval process, the infimum over all schedulers is taken and it is obviously smaller than all the concrete probabilities in the CTMC (the supremum coincides with the probabilities for s_1). A detailed discussion on which *schedulers* yield the infimum or supremum is given in the next section.

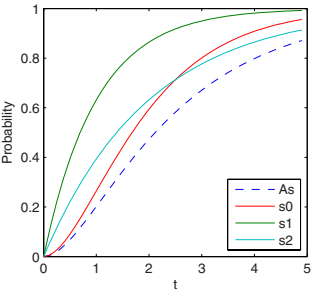


Fig. 4. Concrete vs. abstract behavior over time

Definition 5 (k-step forward simulation). *Let $\mathcal{C} = (S_\mathcal{C}, \mathbf{P}, \lambda, L_\mathcal{C}, s_\mathcal{C})$ be a CTMC and $\mathcal{E} = (S_\mathcal{E}, \mathbf{P}_l, \mathbf{P}_u, \lambda, k, L_\mathcal{E}, s_\mathcal{E})$ an Erlang-k interval process. Relation $\mathcal{R}_k \subseteq S_\mathcal{C} \times S_\mathcal{E}$ is a* k-step forward simulation *on $\mathcal{C}$ and $\mathcal{E}$ iff for all $s \in S_\mathcal{C}$, $s' \in S_\mathcal{E}$, $s\mathcal{R}_k s'$ implies:*

1. *Let $\mu := \mathbf{P}^k(s, \cdot)$. Then there exists $\mu' \in \mathbf{T}_\mathcal{E}(s')$ and $\Delta : S_\mathcal{C} \times S_\mathcal{E} \to [0, 1]$ s.t.*
   (a) $\Delta(u, v) > 0 \Rightarrow u\mathcal{R}_k v$, (b) $\Delta(u, S_\mathcal{E}) = \mu(u)$, (c) $\Delta(S_\mathcal{C}, v) = \mu'(v)$.
2. *For all $a \in AP$, $L_\mathcal{E}(s', a) \neq ?$ implies that $L_\mathcal{E}(s', a) = L_\mathcal{C}(s, a)$.*

We write $s \preceq_k s'$ if $s\mathcal{R}_k s'$ for some k-step forward simulation $\mathcal{R}_k$, and $\mathcal{C} \preceq_k \mathcal{E}$ if $s_\mathcal{C}\mathcal{R}_k s_\mathcal{E}$. In the sequel, we often omit subscript k. The main difference with existing simulation relations is that k steps in $\mathcal{C}$ are matched with a single step in $\mathcal{E}$. For k=1, our definition coincides with the standard notion of forward simulation on CTMCs [4].

Theorem 2 (Abstraction). *Let $\mathcal{C}$ be a CTMC and let $\mathcal{A}$ be a partitioning on the state space S. Then for all $k \in \mathbb{N}^+$ we have $\mathcal{C} \preceq$ abstr$(\mathcal{C}, \mathcal{A}, k)$.*

It is important to understand that the k-step forward simulation relates the transition probabilities of one transition in the abstract system to k-transitions in the concrete system. However, it does not say anything about the number $j \in \{1, \ldots, k\}$ of arrivals in the Poisson process, which has to be chosen appropriately to guarantee that the probability for reaching certain states within a given time bound is related in the concrete and the abstract system. This issue will be approached in the next section.

5 Reachability

We now show that the abstraction method proposed above can be used to efficiently derive bounds for the probability to reach a set $B \subseteq S_\mathcal{C}$ in a CTMC $\mathcal{C} = (S_\mathcal{C}, \mathbf{P}, \lambda, L_\mathcal{C}, s_\mathcal{C})$.

For that we consider an Erlang-k interval process $\mathcal{E}$ with state space $S_{\mathcal{E}}$ and $\mathcal{C} \preceq \mathcal{E}$. For $B' \subseteq S_{\mathcal{E}}$, $t \geq 0$ let $Reach_{\leq t}(B') := \{\sigma \in Path_{\mathcal{E}} \mid \exists t' \in [0,t] : \sigma@t' \in B'\}$.

Since a CTMC is also an Erlang-k interval process, $Reach_{\leq t}(B) \subseteq Path_{\mathcal{C}}$ is defined in the same way. We assume that $\mathbf{P}(s,s) = 1$ for all $s \in B$ as the behavior of $\mathcal{C}$ after visiting B can be ignored. We say that B and B' are *compatible* iff $s \preceq s'$ implies that $s \in B$ iff $s' \in B'$, for all $s \in S_{\mathcal{C}}$, $s' \in S_{\mathcal{E}}$. For example, in Fig. 4, $B = \{u\}$ and $B' = \{A_u\}$, as well as, $B = \{s_0, s_1, s_2\}$ and $B' = \{A_s\}$ are compatible.

The k-step forward simulation (cf. Def. 5) is useful for relating transition probabilities in the concrete and the abstract system. However, to relate *timed* reachability probabilities of concrete and abstract systems, we have to assess the time abstract transitions with the *right* number j of new arrivals in the Poisson process associated with $\mathcal{E}$. In other words, we have to check for which choice of the number of arrivals, we obtain lower and upper bounds of the timed reachability probabilities. As motivated in the introduction (Fig. 2) and stated in Theorem 3 (see below), a tight bound for

- the minimum probability is obtained when the scheduler chooses for number j always k, and a tight bound for
- the maximum probability is obtained when the scheduler chooses once $j = 1$ and for the remaining transitions $j = k$.

Consequently, we restrict our attention to the following scheduler classes:

$$\mathcal{HD}_l^{\mathcal{E}} := \{D \in \mathcal{HD}^{\mathcal{E}} \mid \forall \xi \exists \mu_\xi : D(\xi) = (\mu_\xi, k)\}$$
$$\mathcal{HD}_u^{\mathcal{E}} := \{D \in \mathcal{HD}^{\mathcal{E}} \mid \forall \xi \exists \mu_\xi : D(\xi) = (\mu_\xi, 1) \text{ if } \xi = s_{\mathcal{E}},\ D(\xi) = (\mu_\xi, k) \text{ otherwise}\}$$

where $s_{\mathcal{E}}$ is the initial state of the Erlang-k interval process $\mathcal{E}$.

Theorem 3. *Let $\mathcal{C}$ be a CTMC and $\mathcal{E}$ an Erlang-k interval process with $\mathcal{C} \preceq \mathcal{E}$. For $t \in \mathbb{R}_{\geq 0}$, compatible sets B and B', there exist schedulers $D \in \mathcal{HD}_l^{\mathcal{E}}$, $D' \in \mathcal{HD}_u^{\mathcal{E}}$ with*

$$Pr^{D}(Reach_{\leq t}(B')) \leq Pr^{\mathcal{C}}(Reach_{\leq t}(B)) \leq Pr^{D'}(Reach_{\leq t}(B')).$$

Let

$$Pr_l^{\mathcal{E}}(Reach_{\leq t}(B')) := \inf_{D \in \mathcal{HD}_l^{\mathcal{E}}} Pr^{D}(Reach_{\leq t}(B'))$$
$$Pr_u^{\mathcal{E}}(Reach_{\leq t}(B')) := \sup_{D \in \mathcal{HD}_u^{\mathcal{E}}} Pr^{D}(Reach_{\leq t}(B')).$$

The following corollary is a direct result of the theorem above. It states that when comparing reachability probabilities of a CTMC with those of a simulating Erlang-k interval process $\mathcal{E}$, in the *worst (best) case* $\mathcal{E}$ will have a smaller (larger) time-bounded reachability probability, when restricting to the scheduler class $\mathcal{HD}_l^{\mathcal{E}}$ ($\mathcal{HD}_u^{\mathcal{E}}$).

Corollary 1. *Let $\mathcal{C}$ be a CTMC and $\mathcal{E}$ an Erlang-k interval process with $\mathcal{C} \preceq \mathcal{E}$. Let $t \in \mathbb{R}_{\geq 0}$ and B be compatible with B'. Then:*

$$Pr_l^{\mathcal{E}}(Reach_{\leq t}(B')) \leq Pr^{\mathcal{C}}(Reach_{\leq t}(B)) \leq Pr_u^{\mathcal{E}}(Reach_{\leq t}(B'))$$

Similar to the uniformization method for CTMCs (see Section 2), we can efficiently calculate time-bounded reachability probabilities in $\mathcal{E}$, using time-abstract reachability probabilities and the probability for the number of Poisson arrivals in a certain range.

More specifically, after i transitions in $\mathcal{E}$, the number of arrivals in the associated Poisson process is among $i\cdot k, i\cdot k{+}1, \ldots, i\cdot k{+}(k{-}1)$, if $D \in \mathcal{HD}_l^{\mathcal{E}}$, and $(i{-}1)\cdot k{+}1, (i{-}1)\cdot k{+}2, \ldots, i\cdot k$, if $D \in \mathcal{HD}_u^{\mathcal{E}}$. For $B \subseteq S_{\mathcal{E}}$, $i \in \mathbb{N}$ let $Reach^{=i}(B) := \{\sigma \in Path_{\mathcal{E}} \mid \sigma[i] \in B\}$. Using $\psi_{\lambda,t}$ for the respective Poisson probabilities, we thus obtain:

Lemma 2. *Let $\mathcal{E}$ be an Erlang-k interval process, $t \in \mathbb{R}_{\geq 0}$ and $B \subseteq S_{\mathcal{E}}$. Then*

$$Pr^D(Reach_{\leq t}(B)) = \sum_{i=0}^{\infty} \left(Pr^D_{ta}(Reach^{=i}(B)) \cdot \psi_{\lambda,t}(\textstyle\sum_{h=0}^{i-1} j_h, j_i) \right)$$

where $j_i = k$ for all $i \in \mathbb{N}$ if $D \in \mathcal{HD}_l^{\mathcal{E}}$ and $j_0 = 1$, $j_i = k$ for $i \in \mathbb{N}^+$ if $D \in \mathcal{HD}_u^{\mathcal{E}}$.

Similar as in [2], we can approximate the supremum/infimum w.r.t. the scheduler classes $\mathcal{HD}_l^{\mathcal{E}}$ and $\mathcal{HD}_u^{\mathcal{E}}$ by applying a greedy strategy for the optimal choices of distributions $\mathbf{P} \in \mathbf{T}_{\mathcal{E}}$. A truncated, step-dependent scheduler is sufficient to achieve an accuracy of $1-\epsilon$ where the error bound $\epsilon > 0$ is specified a priori. The decisions of this scheduler only depend on the number of transitions performed so far and its first $N := N(\epsilon)$ decisions can be represented by a sequence $\mathbf{P}_1, \ldots, \mathbf{P}_N \in \mathbf{T}_{\mathcal{E}}$. As discussed in Section 3, it suffices if the matrices are such that only extreme distributions are involved. As the principle for the greedy algorithm is similar for suprema and infima, we focus on the former. Let i_B be the vector of size $|S_{\mathcal{E}}|$ with $\mathsf{i}_B(s) = 1$ iff $s \in B$. Furthermore, $\mathbf{P}_0 := \mathbf{I}$ and $\mathsf{v}_i := \prod_{m=0}^{i} \mathbf{P}_m \cdot \mathsf{i}_B$. We choose matrices $\mathbf{P}_i$, $i \geq 1$ such that

$$|Pr^{\mathcal{E}}_u(Reach_{\leq t}(B)) - \textstyle\sum_{i=0}^{N} \mathsf{v}_i(s_{\mathcal{E}}) \cdot \psi_{\lambda,t}(\sum_{h=0}^{i-1} j_h, j_i)| < \epsilon.$$

The algorithm is illustrated in Fig. 5 and has polynomial time complexity. Starting in a backward manner, i.e., with $\mathbf{P}_N$, vector q_i^u is maximized by successively assigning as much proportion as possible to the transition leading to the successor s' for which $q_{i+1}^u(s')$ is maximal. For every choice of a value $\mathbf{P}_i(s,s')$ the transition probability intervals for the remaining choices are normalized (compare Example 1). Note that the algorithm computes bounds which may be with an error bound ϵ *below* the actual value. Thus, the computed lower bound may be lower than the actual lower bound. To assure that the upper bound exceeds the actual upper bound, we add ϵ to q_0^u.

The following lemma is an adaptation of [2, Th. 5] and states that the results are indeed ϵ-approximations of the supremum/infimum of the reachability probabilities.

Input: Erlang-k interval process $\mathcal{E}$, time bound t, set of states B	Input: Erlang-k interval process $\mathcal{E}$, time bound t, set of states B
Output: ϵ-approx. q_0^l of $Pr_l^{\mathcal{E}}(Reach_{\leq t}(B))$	Output: ϵ-approx. q_0^u of $Pr_u^{\mathcal{E}}(Reach_{\leq t}(B))$
Minimize q_0^l where for $1 \leq i \leq N$	*Maximize q_0^u where for* $1 \leq i \leq N$
$q_0^l = \psi_{\lambda,t}(0,k)\,\mathsf{i}_B + q_1^l$	$q_0^u = \psi_{\lambda,t}(0,1)\,\mathsf{i}_B + q_1^u + \epsilon$
$q_i^l = \psi_{\lambda,t}(ik,k)\,\mathbf{P}_i\mathsf{i}_B + \mathbf{P}_i\, q_{i+1}^l$	$q_i^u = \psi_{\lambda,t}(1+(i{-}1)k,k)\,\mathbf{P}_i\mathsf{i}_B + \mathbf{P}_i\, q_{i+1}^u$
$q_{N+1}^l = \underline{0}$	$q_{N+1}^u = \underline{0}$

Fig. 5. Greedy algorithm for infimum (left) and supremum (right) of time-bounded reachability probabilities

Table 2. Three-valued semantics of CSL

$$[\![\mathit{true}]\!](s) = \top \qquad [\![a]\!](s) = L(s,a)$$
$$[\![\varphi_1 \wedge \varphi_2]\!](s) = [\![\varphi_1]\!](s) \sqcap [\![\varphi_2]\!](s) \qquad [\![\neg\varphi]\!](s) = ([\![\varphi]\!](s))^c$$

$$[\![\varphi_1 \mathcal{U}^I \varphi_2]\!](\sigma) = \begin{cases} \top & \text{if } \exists\, t \in I : ([\![\varphi_2]\!](\sigma@t) = \top \wedge\ \forall\, t' \in [0,t) : [\![\varphi_1]\!](\sigma@t') = \top) \\ \bot & \text{if } \forall\, t \in I : ([\![\varphi_2]\!](\sigma@t) = \bot \vee\ \exists\, t' \in [0,t) : [\![\varphi_1]\!](\sigma@t') = \bot) \\ ? & \text{otherwise} \end{cases}$$

$$[\![\mathcal{P}_{\trianglerighteq p}(\varphi_1 \mathcal{U}^I \varphi_2)]\!](s) = \begin{cases} \top & \text{if } \mathit{Pr}_l(s, \varphi_1 \mathcal{U}^I \varphi_2) \trianglerighteq p \\ \bot & \text{if } \mathit{Pr}_u(s, \varphi_1 \mathcal{U}^I \varphi_2) \triangleleft p \\ ? & \text{otherwise} \end{cases} \qquad \trianglerighteq \in \{>, \geq\}, \triangleleft = \begin{cases} < & \text{if } \trianglerighteq = \leq \\ \leq & \text{if } \trianglerighteq = < \end{cases}$$

$$[\![\mathcal{P}_{\trianglelefteq p}(\varphi_1 \mathcal{U}^I \varphi_2)]\!](s) = \begin{cases} \top & \text{if } \mathit{Pr}_u(s, \varphi_1 \mathcal{U}^I \varphi_2) \trianglelefteq p \\ \bot & \text{if } \mathit{Pr}_l(s, \varphi_1 \mathcal{U}^I \varphi_2) \triangleright p \\ ? & \text{otherwise} \end{cases} \qquad \trianglelefteq \in \{<, \leq\}, \triangleright = \begin{cases} > & \text{if } \trianglelefteq = \geq \\ \geq & \text{if } \trianglelefteq = > \end{cases}$$

Lemma 3. *For an Erlang-k interval process $\mathcal{E}$, $B \subseteq S_{\mathcal{E}}$, $t \geq 0$, error margin $\epsilon > 0$:*

$$\mathit{Pr}_l^{\mathcal{E}}(\mathit{Reach}_{\leq t}(B)) \geq q_0^l(s_{\mathcal{E}}) \geq \mathit{Pr}_l^{\mathcal{E}}(\mathit{Reach}_{\leq t}(B)) - \epsilon$$
$$\mathit{Pr}_u^{\mathcal{E}}(\mathit{Reach}_{\leq t}(B)) \leq q_0^u(s_{\mathcal{E}}) \leq \mathit{Pr}_u^{\mathcal{E}}(\mathit{Reach}_{\leq t}(B)) + \epsilon.$$

We conclude this section with a result that allows us to use the algorithm presented above to check if a reachability probability is at least (at most) p in the abstract model and, in case the result is positive, to deduce that the same holds in the concrete model.

Theorem 4. *For a CTMC $\mathcal{C}$, an Erlang-k interval process $\mathcal{E}$ with $\mathcal{C} \preceq \mathcal{E}$, compatible sets $B \subseteq S_{\mathcal{C}}$, $B' \subseteq S_{\mathcal{E}}$, $t \geq 0$, $\epsilon > 0$, the algorithm in Fig. 5 computes q_0^l and q_0^u with:*

$$\mathit{Pr}^{\mathcal{C}}(\mathit{Reach}_{\leq t}(B)) \geq \mathit{Pr}_l^{\mathcal{E}}(\mathit{Reach}_{\leq t}(B')) \geq q_0^l(s_{\mathcal{E}}) \geq \mathit{Pr}_l^{\mathcal{E}}(\mathit{Reach}_{\leq t}(B')) - \epsilon$$
$$\mathit{Pr}^{\mathcal{C}}(\mathit{Reach}_{\leq t}(B)) \leq \mathit{Pr}_u^{\mathcal{E}}(\mathit{Reach}_{\leq t}(B')) \leq q_0^u(s_{\mathcal{E}}) \leq \mathit{Pr}_u^{\mathcal{E}}(\mathit{Reach}_{\leq t}(B')) + \epsilon.$$

6 Model Checking

The characterizations in Section 5 in terms of minimal and maximal time-bounded reachability probabilities are now employed for model checking CSL on Erlang-k interval processes. Therefore, we define a three-valued CSL semantics and show that verification results on Erlang-k interval processes carry over to their underlying CTMCs.

Three-valued semantics. For Erlang-k interval process $\mathcal{E} = (S, \mathbf{P}_l, \mathbf{P}_u, \lambda, k, L, s_0)$, we define the satisfaction function $[\![\cdot]\!] : \text{CSL} \to (S \cup \mathit{Path}_{\mathcal{E}} \to \mathbb{B}_3)$ as in Table 2, where $s \in S$, $\mathcal{E}_s$ is defined as $\mathcal{E}$ but with initial state s and

$$\mathit{Pr}_l(s, \varphi_1 \mathcal{U}^I \varphi_2) = \mathit{Pr}_l^{\mathcal{E}_s}(\{\sigma \in \mathit{Path}_{\mathcal{E}_s} \mid [\![\varphi_1 \mathcal{U}^I \varphi_2]\!](\sigma) = \top\}) \tag{1}$$
$$\mathit{Pr}_u(s, \varphi_1 \mathcal{U}^I \varphi_2) = \mathit{Pr}_u^{\mathcal{E}_s}(\{\sigma \in \mathit{Path}_{\mathcal{E}_s} \mid [\![\varphi_1 \mathcal{U}^I \varphi_2]\!](\sigma) \neq \bot\}) \tag{2}$$

For the propositional fragment the semantics is clear. A path σ satisfies until formula $\varphi_1 \mathcal{U}^{[0,t]} \varphi_2$ if φ_1 definitely holds until φ_2 definitely holds at the latest at time t. The until-formula is violated, if either before φ_2 holds, φ_1 is violated, or if φ_2 is definitely

violated up to time t. Otherwise, the result is indefinite. To determine the semantics of $\mathcal{P}_{\leq p}(\varphi_1\, \mathcal{U}^{[0,t]} \varphi_2)$, we consider the probability of the paths for which $\varphi_1\, \mathcal{U}^{[0,t]} \varphi_2$ is definitely satisfied or perhaps satisfied, i.e., indefinite. If this probability is at most p then $\mathcal{P}_{\leq p}(\varphi_1\, \mathcal{U}^{[0,t]} \varphi_2)$ is definitely satisfied. Similarly, $\mathcal{P}_{\leq p}(\varphi_1\, \mathcal{U}^{[0,t]} \varphi_2)$ is definitely violated if this probability exceeds p for those paths on which $\varphi_1\, \mathcal{U}^{[0,t]} \varphi_2$ evaluates to $\top$. The semantics of $\mathcal{P}_{\trianglelefteq p}(\varphi_1\, \mathcal{U}^{[0,t]} \varphi_2)$ for $\trianglelefteq \in \{<, >, \geq\}$ follows by a similar argumentation.

Theorem 5 (Preservation). *For a CTMC $\mathcal{C}$ and an Erlang-k interval process $\mathcal{E}$ with initial states $s_{\mathcal{C}}$ and $s_{\mathcal{E}}$, if $s_{\mathcal{C}} \preceq s_{\mathcal{E}}$ then for all CSL formulas φ:*

$$[\![\varphi]\!](s_{\mathcal{E}}) \neq ?\ \textit{implies}\ [\![\varphi]\!](s_{\mathcal{E}}) = [\![\varphi]\!](s_{\mathcal{C}}).$$

Model checking three-valued CSL is, as usual, done bottom-up the parse tree of the formula. The main task is checking until-subformulas $\mathcal{P}_{\leq p}(a\, \mathcal{U}^{[0,t]} b)$, which can be handled as follows: As in [7], the underlying transition system is transformed such that there is one sink for all states satisfying b and another one for all states neither satisfying a nor b. Thus, all paths reaching states satisfying b are along paths satisfying a, which allows to compute the measure for reaching b states. However, before doing so, we have to account for indefinite states ($?$): When computing lower bounds we consider all states labeled by $?$ as ones labeled $\bot$, while we consider them as labeled $\top$ when computing upper bounds, following equations (1) and (2).

Example 3. Consider Ex. 2 where state u (and thus A_u) are labeled *goal*, and CSL formula $\varphi = \mathcal{P}_{\leq 0.9}(\mathit{true}\, \mathcal{U}^{\leq 1.2} \mathit{goal})$. Then $[\![\varphi]\!](A_s) = \top = [\![\varphi]\!](s_0)$ (compare Fig. 4). If s_1 was labeled *goal* as well then $L(A_s, \mathit{goal}) = ?$. Checking φ for satisfaction requires an optimistic relabeling, i.e. we set $L(A_s, \mathit{goal}) = \top$. Obviously, then φ is not satisfied for sure. Analyzing the pessimistic instance with $L(A_s, \mathit{goal}) = \bot$ however yields that φ is neither violated for sure (cf. Fig. 4). Therefore $[\![\varphi]\!](A_s) = ?$ implying that either the partitioning or the choice of k has to be revised in order to get conclusive results.

Theorem 6 (Complexity). *Given an Erlang-k interval process $\mathcal{E}$, a CSL formula φ, and an error margin ϵ, we can approximate $[\![\varphi]\!]$ in time polynomial in the size of $\mathcal{E}$ and linear in the size of φ, the exit rate λ and the highest time bound t occurring in φ (dependency on ϵ is omitted as ϵ is linear in λt). In case the approximation yields $\top$ or $\bot$, the result is correct.*

7 Case Study: Enzymatic Reaction

Markovian models are well established for the analysis of biochemical reaction networks [5,15]. Typically, such networks are described by a set of reaction types and the involved molecular species, e.g., the different types of molecules. The occurrence of a reaction changes the species' populations as molecules are produced and/or consumed.

Enzyme-catalyzed substrate conversion. We focus on an enzymatic reaction network with four molecular species: enzyme (E), substrate (S), complex (C) and product (P) molecules. The three reaction types R_1, R_2, R_3 are given by the following rules:

$$R_1 : E + S \xrightarrow{c_1} C, \quad R_2 : C \xrightarrow{c_2} E + S, \quad R_3 : C \xrightarrow{c_3} E + P$$

The species on the left hand of the arrow (also called *reactants*) describe how many molecules of a certain type are consumed by the reaction and those on the right hand describe how many are produced. For instance, one molecule of type E and S is consumed by reaction R_1 and one C molecule is produced. The constants $c_1, c_2, c_3 \in \mathbb{R}_{>0}$ determine the probability of the reactions as explained below.

Concrete model. The temporal evolution of the system is represented by a CTMC as follows (cf. [6]): A state corresponds to a population vector $x = (x_E, x_S, x_C, x_P) \in \mathbb{N}^4$ and transitions are triggered by chemical reactions. The change of the current population vector x caused by a reaction of type R_m, $m \in \{1, 2, 3\}$ is expressed as a vector v_m where $v_1 := (-1, -1, 1, 0)$, $v_2 := (1, 1, -1, 0)$ and $v_3 := (1, 0, -1, 1)$. Obviously, reaction R_m is only possible if vector $x + v_m$ contains no negative entries. Given an initial state $s := (s_E, s_S, 0, 0)$, it is easy to verify that the set of reachable states equals $S := \{(x_E, x_S, x_C, x_P) \mid x_E + x_C = s_E, x_S + x_C + x_P = s_S\}$.

The probability that a reaction of type R_m occurs within a certain time interval is determined by the function $\alpha_m : S \to \mathbb{R}_{\geq 0}$. The value $\alpha_m(x)$ is proportional to the number of distinct combinations of R_m's reactants: $\alpha_1(x) := c_1 x_E x_S$, $\alpha_2(x) := c_2 x_C$ and $\alpha_3(x) := c_3 x_C$. We define the transition matrix $\mathbf{P}$ of the CTMC by $\mathbf{P}(x, x + v_m) := \alpha_m(x)/\lambda$ with exit rate $\lambda \geq \max_{x \in S}(\alpha_1(x) + \alpha_2(x) + \alpha_3(x))$. Thus, state x has outgoing transitions $x \xrightarrow{\alpha_m(x)/\lambda} x + v_m$ for all m with $x + v_m \geq \underline{0}$ and the self-loop probability in x is $\mathbf{P}(x, x) := 1 - \big(\alpha_1(x) + \alpha_2(x) + \alpha_3(x)\big)/\lambda$.

We are interested in the probability that within time t the number of type P molecules reaches threshold $n := s_S$, the maximum number of P molecules. We apply labels $AP := \{0, 1, \ldots, n\}$ and for $0 \leq a \leq n$ let $L(x, a) := \top$ if $x = (x_E, x_S, x_C, x_P)$ with $x_P = a$ and $L(x, a) := \bot$ otherwise. For the initial populations, we fix $s_E = 20$ and vary s_S between 50 and 2000.

Stiffness. In many biological systems, components act on time scales that differ by several orders of magnitude which leads to *stiff* models. Traditional numerical analysis methods perform poorly in the presence of stiffness because a large number of very small time steps has to be considered. For the enzymatic reaction, stiffness arises if $c_2 \gg c_3$ and results in a high self-loop probability in most states because λ is large compared to $\alpha_1(x) + \alpha_2(x) + \alpha_3(x)$. Thus, even in case of a small number $|S|$ of reachable states, model checking properties like $\mathcal{P}_{\leq 0.9}(\mathit{true}\, \mathcal{U}^{[0,t]} n)$ is extremely time consuming. We show how our abstraction method can be used to efficiently verify properties of interest even for stiff parameter sets. We choose a realistic parameter set of $c_1 = c_2 = 1$ and $c_3 = 0.001$. Note that the order of magnitude of the expected time until threshold $n = s_S = 300$ is reached is 10^4 for these parameters.

Abstract model. For the CTMC $\mathcal{C} := (S, \mathbf{P}, \lambda, L, s)$ described above, we choose partitioning $\mathcal{A} := \{A_0, \ldots, A_n\}$ with $A_a := \big\{x \in S \mid L(x, a) = \top\big\}$, that is, we group all states in which the number of molecules of type P is the same. Some important remarks are necessary at this point. Abstraction techniques rely on the construction of small abstract models by disregarding details of the concrete model as the latter is too large to be solved efficiently. In this example, we have the additional problem of stiffness and the abstraction method proposed here can tackle this by choosing high values for k. Then one step in the Erlang-k interval process happens after a large number of

arrivals in the underlying Poisson process and the self-loop probability in the abstract model is much smaller than in the concrete one. We chose $k \in \{2^{10}, 2^{11}, 2^{12}\}$ for the construction of the Erlang-k interval process *abstr*$(\mathcal{C}, \mathcal{A}, k)$ and calculate the transition probability intervals by taking the k-th matrix power of $\mathbf{P}$. The choice for k is reasonable, since for a given error bound $\epsilon = 10^{-10}$, $s_S = 300$ and $t = 10000$, a transient analysis of the concrete model via uniformization would require around $6 \cdot 10^7$ steps. By contrast, our method considers k steps in the concrete model and around $(6 \cdot 10^7)/k$ steps in the smaller abstract model. Thus, although the construction of the Erlang-k interval process is expensive, the total time savings are enormous. We used the MATLAB software for our prototypical implementation and the calculation of $\mathbf{P}^k$ could be performed efficiently because $\mathbf{P}^{2^j}$ can be computed using j matrix multiplications. As for non-stiff models a smaller value is chosen for k, it is obvious that upper and lower bounds for the k-step transition probabilities can be obtained in a local fashion, i.e. by computing the k-th matrix power of submatrices of $\mathbf{P}$. Therefore, we expect our method to perform well even if $|S|$ is large. However, for stiff *and* large concrete models more sophisticated techniques for the construction of the abstract model must be applied that exploit the fact that only upper and lower bounds are needed.

Experimental results. For $s_S = 200$ we compared the results of our abstraction method for the probability to reach A_n within time bound t with results for the concrete model that were obtained using PRISM. While it took more than one day to generate the plot for the concrete model in Fig. 7, right, our MATLAB implementation took less than one hour for all three pairs of upper and lower probability bounds and different values of t.[2] Our method is accurate as the obtained intervals are small, e.g., for $s_S = 200$, $k = 2^{12}$, $t = 14000$ the relative interval width is

$\lvert\mathcal{A}\rvert$	$\lvert S\rvert$	time
50	861	$0m\ \ 5s$
300	6111	$37m\ 36s$
500	10311	$70m\ 39s$
1000	20811	$144m\ 49s$
1500	31311	$214m\ \ 2s$
2000	41811	$322m\ 50s$

Fig. 6. Computation times

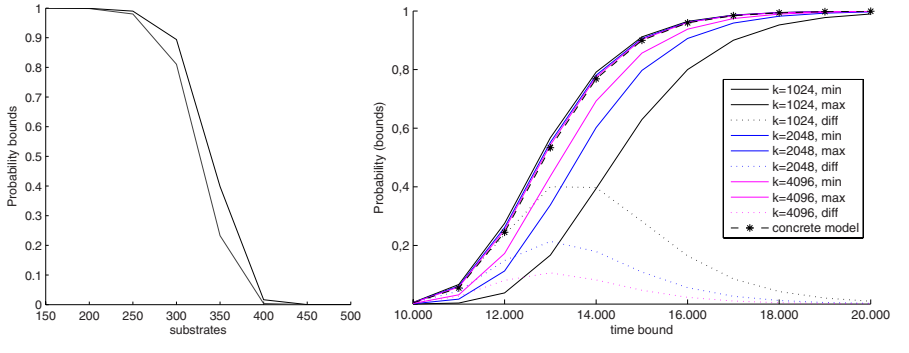


Fig. 7. Time-bounded reachability

[2] Both jobs were run on the same desktop computer (Athlon 64 X2 3800+, 2GB RAM).

10.7%. Fig. 7, left, shows the lower and upper probability bounds using $k = 2^{12}$, $t = 20000$ and varying s_S. For high values of s_S, e.g., $s_S = 500$ the construction of the Erlang-k interval process took more than 99% of the total computation time as the size of the transition matrix $\mathbf{P}$ is $10^4 \times 10^4$ and sparsity is lost during matrix multiplication. We conclude this section with the additional experimental details on computation times[3], given in Fig. 6, using $k = 2^{12}$, $t = 50000$ (and $s_S = 200$).

Note that for this case study exact abstraction techniques such as lumping do not yield any state-space reduction.

8 Conclusion

We have presented an abstraction technique for model checking of CTMCs, presented its theoretical underpinnings, as well as an the application of the abstraction technique to a well-known case study from biochemistry. The main novel aspect of our approach is that besides the abstraction of transition probabilities by intervals [10,17], sequences of transitions may be collapsed yielding an approximation of the timing behavior. Abstract Erlang k-interval processes are shown to provide under- and overapproximations of time-bounded reachability probabilities. Our case study confirms that these bounds may be rather accurate. Future work will focus on automatically finding suitable state-space partitionings, and on guidelines for selecting k appropriately. As shown by our case study, for stiff CTMCs, a high value of k is appropriate. This is, however, not the case in general. We anticipate that graph analysis could be helpful to select a "good" value for k. Moreover, we plan to investigate memory-efficient techniques for computing k-step transition probabilities and counterexample-guided abstraction refinement.

References

1. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous time Markov chains. ACM TOCL 1, 162–170 (2000)
2. Baier, C., Hermanns, H., Katoen, J.P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. TCS 345, 2–26 (2005)
3. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. IEEE TSE 29, 524–541 (2003)
4. Baier, C., Katoen, J.-P., Hermanns, H., Wolf, V.: Comparative branching-time semantics for Markov chains. Information and Computation 200, 149–214 (2005)
5. Bower, J.M., Bolouri, H.: Computational Modeling of Genetic and Biochemical Networks. MIT Press, Cambridge (2001)
6. Busch, H., Sandmann, W., Wolf, V.: A numerical aggregation algorithm for the enzyme-catalyzed substrate conversion. In: Priami, C. (ed.) CMSB 2006. LNCS (LNBI), vol. 4210, pp. 298–311. Springer, Heidelberg (2006)
7. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42, 857–907 (1995)

[3] Run on a workstation (Xeon 5140 – 2.33 GHz, 32GB RAM).

8. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001)
9. de Alfaro, L., Pritam, R.: Magnifying-lens abstraction for Markov decision processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 325–338. Springer, Heidelberg (2007)
10. 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)
11. Feller, W.: An Introduction to Probability Theory and its Applications, vol. I. John Wiley & Sons, Inc., Chichester (1968)
12. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6, 512–535 (1994)
13. Hermanns, H., Herzog, U., Katoen, J.-P.: Process algebra for performance evaluation. TCS 274, 43–87 (2002)
14. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123. Springer, Heidelberg (2008)
15. Kampen, N.G.v.: Stochastic Processes in Physics and Chemistry, 3rd edn. Elsevier, Amsterdam (2007)
16. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation minimisation mostly speeds up probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–102. Springer, Heidelberg (2007)
17. Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for continuous-time Markov chains. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 316–329. Springer, Heidelberg (2007)
18. Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Abstraction for stochastic systems by Erlang's method of stages. Technical Report AIB-2008-12, RWTH Aachen University (2008)
19. Kwiatkowska, M., Norman, G., Parker, D.: Game-based abstraction for Markov decision processes. In: QEST, pp. 157–166. IEEE CS Press, Los Alamitos (2006)
20. Kwiatkowska, M., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 234–248. Springer, Heidelberg (2006)
21. Mamoun, M.B., Pekergin, N., Younes, S.: Model checking of CTMCs by closed-form bounding distributions. In: QEST, pp. 189–199. IEEE CS Press, Los Alamitos (2006)
22. Remke, A., Haverkort, B., Cloth, L.: CSL model checking algorithms for QBDs. TCS 382, 24–41 (2007)
23. Ross, S.: Stochastic Processes. John Wiley and Sons, Chichester (1996)
24. Sen, K., Viswanathan, M., Agha, G.: Model-checking Markov chains in the presence of uncertainties. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 394–410. Springer, Heidelberg (2006)
25. Stewart, W.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton (1995)
26. Wolf, V., Baier, C., Majster-Cederbaum, M.: Trace machines for observing continuous-time Markov chains. ENTCS 153, 259–277 (2004)
27. Younes, H., Simmons, R.: Statistical probabilistic model checking with a focus on time-bounded properties. Inf. and Comp. 204, 1368–1409 (2007)
28. Zhang, L., Hermanns, H., Eisenbrand, F., Jansen, D.N.: Flow faster: efficient decision algorithms for probabilistic simulations. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 155–170. Springer, Heidelberg (2007)

On the Minimisation of Acyclic Models*

Pepijn Crouzen[1], Holger Hermanns[1,2], and Lijun Zhang[1]

[1] Universität des Saarlandes, Saarbrücken, Germany
[2] INRIA, VASY, Grenoble Rhône-Alpes, France
{crouzen,hermanns,zhang}@cs.uni-sb.de

Abstract. This paper presents a novel algorithm to compute weak bisimulation quotients for finite acyclic models. It is developed in the setting of interactive Markov chains, a model overarching both labelled transition systems and continuous-time Markov chains. This model has lately been used to give an acyclic compositional semantics to dynamic fault trees, a reliability modelling formalism.

While the theoretical complexity does not change substantially, the algorithm performs very well in practice, almost linear in the size of the input model. We use a number of case studies to show that it is vastly more efficient than the standard bisimulation minimisation algorithms. In particular we show the effectiveness in the analysis of dynamic fault trees.

1 Introduction

Determining the minimum bisimulation quotient of a behavioural model is one of the principal algorithmic challenges in concurrency theory, with concrete applications in many areas. Together with substitutivity properties enjoyed by process algebraic composition operators, bisimulation is at the heart of *compositional aggregation*, one of the most elegant ways to alleviate the state space explosion problem: In compositional aggregation, a model is composed out of sub-models. During generation of its state-space representation, composition and minimisation steps are intertwined along the structure of the compositional specification. This strategy is central to explicit-state verification tools such as μCRL [1] and CADP [12], and arguably a central backbone of their successes in industrial settings [7, 18].

The algorithmic problem to minimise a labelled transition system with respect to bisimulation is well studied. For strong bisimulation, a partition refinement based approach [22] can be used to achieve an algorithm with complexity $\mathcal{O}(m \log n)$, where m and n denote the number of transitions and states of the model. The computation of weak and branching bisimulation is theoretically dominated by the need to compute the transitive closure of internal transitions. This directly determines the overall complexity to be $\mathcal{O}(n^3)$ (disregarding some very specialized algorithms for transitive closure such as [6]). As first noted in [15], the transitive closure computation does *not* dominate in practical applications, and then the complexity is $\mathcal{O}(m^* \log n)$, where m^* is the number of transitions after closure.

* This work is supported by the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS and by the European Commission under the IST framework 7 project QUASIMODO.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 295–309, 2008.

Lately, the growing importance of the minimisation problem has triggered work in at least three different directions. Orzan and Blom have devised an efficient *distributed* algorithm for bisimulation minimisation, based on the notion of *signatures* [2]. Wimmer et al. [25] have taken up this idea to arrive at a fully *symbolic* implementation of the signature-refinement approach, to effectively bridge to BDD-based representations of state spaces. In [9], an algorithm with $\mathcal{O}(m)$ complexity has been proposed for deciding strong bisimulation on directed acyclic graphs. Mateescu [20] developed an $\mathcal{O}(m)$ algorithm for checking modal mu calculus on acyclic LTS, which can be instantiated to checking weak bisimulation, then requiring $\mathcal{O}(m^2)$.

Stochastic behavioural models are among the most prominent application areas for bisimulation minimisation and compositional aggregation techniques [3, 17]. They are used to model and study 'quantified uncertainty" in many areas, such as embedded, networked, randomized, and biological systems. Interactive Markov chains (IMCs) [16] constitute a process algebraic formalism to construct such models. Recently, an extension of IMCs with input and output (IOIMCs) [4] has been introduced to define a rigorous compositional semantics for dynamic fault trees (DFTs). Fault trees and DFTs are in daily use in industrial dependability engineering [10, 24]. The analysis of DFTs via their IOIMC semantics relies heavily on compositional aggregation and weak bisimulation minimisation [4]. Remarkably, the IOIMC semantics maps on acyclic structures. This is the main motivation for the work presented in this paper. We show how to effectively exploit acyclicity of the model in weak bisimulation minimisation. Since (IO)IMCs are a strict superset of LTSs, our results apply to LTSs as well.

The problem of weak bisimulation minimisation on acyclic models is substantially different from the strong bisimulation problem. While not directly developed for LTSs, the rough idea of [9] is to assign to each state a rank which corresponds to the length of the longest path from the state to one of the absorbing states. Observing that *(i)* transitions always move from higher rank states to lower rank states, and *(ii)* only states on the same rank can be bisimilar. allows one to arrive at a linear algorithm. Especially condition *(ii)* is invalid in the weak setting. To overcome this, we use elaborated rank-based techniques to partition the state space on-the-fly during the computation of the weak bisimulation quotient. The resulting algorithm is of linear time complexity in m^*, the size of the weak transition relation. We provide experimental evidence that in practice, the algorithm runs even in linear time in the size of the original relation m. The contributions are developed in the setting of IMCs.

Organisation. The paper is organised as follows. After giving preliminary definitions we discuss in Section 3 how to adapt the strong bisimulation algorithm for acyclic digraphs in [9] to IMCs. Section 4 is devoted to a novel algorithm for weak bisimulation on acyclic IMCs. Section 5 discusses two extensions which allow us to handle the models appearing in DFT analysis. Section 6 presents experimental results after we which we conclude.

Detailed proofs for the theorems in this paper can be found in [8].

2 Preliminaries

In this section we introduce the definition of acyclic interactive Markov chains, strong and weak bisimulations [16].

Definition 1. *An interactive Markov chain (IMC) is a tuple* $\langle S, s^0, A, R_i, R_M \rangle$ *where:* S *is a finite set of states,* $s^0 \in S$ *is the starting state,* A *is a finite set of actions,* $R_i \subseteq S \times A \times S$ *is the set of interactive transitions, and* $R_M \subseteq S \times \mathbb{R}^{>0} \times S$ *is the set of Markovian transitions.*

The label τ is a assumed to be contained in A, it denotes the internal, unobservable action. For $(s, a, t) \in R_i$ we write $s \xrightarrow{a}{}^{i} t$ and for $(s, \lambda, t) \in R_M$ we write $s \xrightarrow{\lambda}{}^{M} t$. Let $R = R_i \cup R_M$. We write $s \xrightarrow{x} s'$ if $(s, x, s') \in R$. In this case x is either an action or a Markovian rate. We write $s \rightarrow^{i} t$ for any interactive transition from s to t and $s \rightarrow^{M} t$ for any such Markovian transition.

States with outgoing τ-transitions are called *unstable*. States without outgoing τ-transitions are called *stable*. We write the reflexive and transitive closure of all internal transitions $s \xrightarrow{\tau}{}^{i} s'$ as $\stackrel{\tau}{\Longrightarrow}$ and say that if $s \stackrel{\tau}{\Longrightarrow} s'$ then s may move internally to state s'. For $s \stackrel{\tau}{\Longrightarrow} s' \xrightarrow{a}{}^{i} s'' \stackrel{\tau}{\Longrightarrow} s'''$ we write $s \stackrel{a}{\Longrightarrow}{}^{i} s'''$. For $s \stackrel{\tau}{\Longrightarrow} s' \xrightarrow{\lambda}{}^{M} s'' \stackrel{\tau}{\Longrightarrow} s'''$ we write $s \stackrel{\lambda}{\Longrightarrow}{}^{M} s'''$. For $s \stackrel{\tau}{\Longrightarrow} s' \xrightarrow{x} s'' \stackrel{\tau}{\Longrightarrow} s'''$ we write $s \stackrel{x}{\Longrightarrow} s'''$.

The cumulative rate from a state s to a set of states C, denoted $\gamma_M(s, C)$, is the sum of the rates of all Markovian transitions from s to states in C: $\gamma_M(s, C) = \sum\{|\lambda \mid (s, \lambda, t) \in R_M \wedge t \in C|\}$, where $\{|\ldots|\}$ denotes a multi-set. The internal backwards closure of a set of states C, denoted C^τ is the set of all states that can reach a state in C via zero or more τ-transitions: $C^\tau = \{s \mid s \stackrel{\tau}{\Longrightarrow} t \wedge t \in C\}$.

A (finite) path π is a sequence $\pi = s_0 x_0 s_1 x_1 \ldots s_n$ in $(S \times (A \cup \mathbb{R}^{>0}))^* \times S$ such that $s_i \xrightarrow{x_i} s_{i+1}$ for all $i = 0, 1, \ldots, n-1$. For a path π we let $first(\pi)$ denote the first state s_0 of π, $last(\pi)$ denote the last state of a finite π, $\pi[i]$ denote the $i+1$-th state s_i of π, $\pi_\alpha[i]$ denote the $i+1$-th label x_i of π, and $len(\pi)$ denote the length n of π. Moreover, we let $wlen(\pi) = \{|\pi_\alpha[i] \mid i = 0, \ldots, len(\pi) - 1 \wedge \pi_\alpha[i] \neq \tau|\}$ denote the weak length of π, which corresponds to the number of observable actions of π. We write $s \overset{\pi}{\rightsquigarrow} s'$ if $first(\pi) = s$ and $last(\pi) = s'$. We write the set of all paths starting from a state s as $Paths(s) = \{\pi \mid \exists s' \in S \cdot s \overset{\pi}{\rightsquigarrow} s'\}$. A path π such that $s \overset{\pi}{\rightsquigarrow} s$ and $len(\pi) > 0$ is called a *cycle*.

The *maximal progress* assumption is usually employed when working with IMCs. It states that if an unobservable (τ) transition is possible in a state, no time may advance prior to taking this (or any other action) transition. In other words, in an unstable state the chance of taking a Markovian transition is given by the probability that the delay associated with the transition is less than or equal to 0. However, this probability is 0, and thus we qualify such a transition as not *plausible*. Semantically, their existence is negligible, which will become apparent in the definition of strong and weak bisimulation. On the other hand, all interactive transitions and all Markovian transitions from stable states are plausible. A path $\pi = s_0 x_0 \ldots s_n$ is plausible if it holds that: for all $i = 0, \ldots, n-1$, if $s_i \xrightarrow{x_i}{}^{M} s_{i+1}$ then s_i is stable. We write the set of all plausible paths starting from a state s as $Paths^P(s)$. A plausible path π from s to t is denoted $s \overset{\pi}{\rightsquigarrow}{}^{P} t$.

Definition 2. *An IMC* $\mathcal{P} = \langle S, s^0, A, R_i, R_M \rangle$ *is* acyclic *if it does not contain any plausible path* π *with* $s \overset{\pi}{\rightsquigarrow}^P s$ *and* $len(\pi) > 0$ *for any* $s \in S$.

An acyclic IMC only contains finite plausible paths since the set of states is by definition finite. We recall the definition of strong and weak bisimulations.

Definition 3. *Let* $\mathcal{P} = \langle S, s^0, A, R_i, R_M \rangle$ *be an IMC. An equivalence relation* $\mathcal{E}$ *on* S *is a strong bisimulation if and only if* $s\mathcal{E}t$ *implies for all* $a \in A$ *and all equivalence classes* C *of* $\mathcal{E}$*, that* $s \xrightarrow{a}{}^i s'$ *implies* $t \xrightarrow{a}{}^i t'$ *with* $s'\mathcal{E}t'$*, and* s *stable implies* $\gamma_M(s, C) = \gamma_M(t, C)$.

Two states s, t *of* $\mathcal{P}$ *are strongly bisimilar, written* $s \sim t$ *if there exists a strong bisimulation* $\mathcal{E}$ *such that* $s\mathcal{E}t$.

Definition 4. *Let* $\mathcal{P} = \langle S, s^0, A, R_i, R_M \rangle$ *be an IMC. An equivalence relation* $\mathcal{E}$ *on* S *is a weak bisimulation if and only if* $s\mathcal{E}t$ *implies for all* $a \in A$ *(including* τ*) and all equivalence classes* C *of* $\mathcal{E}$*, that* $s \overset{a}{\Longrightarrow}{}^i s'$ *implies* $t \overset{a}{\Longrightarrow}{}^i t'$ *with* $s'\mathcal{E}t'$*, and* $s \overset{\tau}{\Longrightarrow} s'$ *and* s' *stable imply* $t \overset{\tau}{\Longrightarrow} t'$ *for some stable* t' *and* $\gamma_M(s', C^\tau) = \gamma_M(t', C^\tau)$.

Two states s, t *of* $\mathcal{P}$ *are weakly bisimilar, written* $s \approx t$ *if there exists a weak bisimulation* $\mathcal{E}$ *such that* $s \approx t$.

Strong, respectively weak bisimilarity is the largest strong, respectively weak bisimulation [16]. For an IMC $\langle S, s^0, A, R_i, \emptyset \rangle$ the above definitions reduce to Milner's standard definitions on labelled transition systems [21].

3 Strong Bisimulation for Acyclic IMCs

To decide strong bisimulation on unlabelled acyclic digraphs, a linear-time algorithm has been developed in [9], which is based on state ranking. To handle labelled transition systems, the authors encode them into unlabeled graphs, for which strong bisimulation can then be decided. In this section, we extend their rank-based algorithm in [9] to decide strong bisimulation directly for IMCs.

We adapt the notion of ranks for acyclic IMCs. The rank of absorbing states is 0, for other states it is the longest distance to a state on rank 0. So the rank of a state is the length of the longest path starting in that state.

Definition 5. *The rank function* $\mathsf{R} : S \to \mathbb{N}$ *is defined by:* $\mathsf{R}(s) = \max\{len(\pi) \mid \pi \in Paths^P(s)\}$.

Since in acyclic IMCs all plausible paths are finite, we have that $\mathsf{R}(s) < \infty$. If state s has a higher rank than state t, we say also that s is higher than t. By definition, transitions always go from higher states to lower states. Before continuing, we state an important observation about the relationship between strong bisimilarity and ranks.

Theorem 1. *If two states of* $\mathcal{P}$ *are strongly bisimilar they are on the same rank:* $\forall s, t \in S \cdot s \sim t \to \mathsf{R}(s) = \mathsf{R}(t)$.

Algorithm 1. Determining the strong bisimilarity quotient for an acyclic IMC

Require: $\mathcal{P} = \langle S, s^0, A, R_i, R_M \rangle$ is an acyclic IMC.

1: $\mathsf{R} = \textsc{ComputeRanks}()$
2: $maxrank = max\{n \mid s \in S \wedge \mathsf{R}(s) = n\}$
3: $BLOCKS = \langle \{\{s \mid s \in S \wedge \mathsf{R}(s) = n\}\} \mid n \leftarrow \langle 0 \ldots maxrank \rangle \rangle$
4: $matrix = \underline{0}$; $rate = \underline{0}$
5: **for** $i = 0$ to $maxrank - 1$ **do**
6: **for all** $(s, a, B) \in \{(s, a, B) \mid B \in BLOCKS[i] \wedge \exists t \in B \cdot (s, a, t) \in R_i\}$ **do**
7: $matrix[s][a][B] = 1$
8: **for all** $(s, \lambda, B) \in \{|(s, \lambda, B) \mid B \in BLOCKS[i] \wedge \exists t \in B \cdot (s, \lambda, t) \in R_M|\}$ **do**
9: **if** s stable **then**
10: $rate[s][B] = rate[s][B] + \lambda$
11: **for** $j = i + 1$ to $maxrank$ **do**
12: $BLOCKS[j] = \bigcup\{\{\{t \mid t \in B \wedge matrix[t] = matrix[s] \wedge rate[t] = rate[s]\} \mid s \in B\} \mid B \in BLOCKS[j]\}$

The above theorem mirrors Proposition 4.2 in [9] and implies that only states on the same rank can be bisimilar. Since transitions go from higher states to lower states, whether two states on the same rank are bisimilar depends only on the states below them. The main idea of the algorithm is to start with the states on rank 0 and to process states rank by rank in a backward manner. The algorithm is presented in Algorithm 1. The input is an acyclic IMC. Lists (and tuples) are written by: $\langle ... \rangle$. The state ranks are computed in line 1 with a simple depth first search (time-complexity $\mathcal{O}(m)$). The partition $BLOCKS$ is initialised such that states with the same rank are grouped together. During the algorithm, we use $BLOCKS[i]$ to denote the partition of the states with rank i. The matrices $matrix$ and $rate$ respectively denote the interactive transitions and the cumulative rates from states to blocks of bisimilar states. The algorithm starts with the rank 0 states which are all strongly bisimilar. Then, it traverses the transitions backwards to refine blocks on the higher level according to the bisimulation definition. Observe that in iteration i all states with ranks lower than i have been processed. Since each transition is visited at most once, the algorithm runs in linear time.

Theorem 2. *Given an acyclic IMC $\mathcal{P}$, Algorithm 1 computes the strong bisimulation correctly. Moreover, the time-complexity of the algorithm is $\mathcal{O}(m)$.*

4 Weak Bisimulation Minimisation

Weak bisimulation (or observational equivalence [21]) differs from strong bisimulation in that only visible behavior has to be mimicked (see Definition 4). In general weak bisimulation can be computed by first computing the reflexive transitive closure of internal transitions $\stackrel{\tau}{\Longrightarrow}$ and then computing the *weak* transitions $s \stackrel{a}{\Longrightarrow} t$ from $s \stackrel{a}{\Longrightarrow} t \leftrightarrow s \stackrel{\tau}{\Longrightarrow} s' \stackrel{a}{\rightarrow} t' \stackrel{\tau}{\Longrightarrow} t$. Once we have computed the weak transition relation we can then compute weak bisimulation simply by computing strong bisimulation on the weak transition relation.

If we try this strategy for acyclic models we quickly run into a problem, since the weak transition relation of an acyclic model is not acyclic, it contains cycles $s \stackrel{\tau}{\Longrightarrow} s$ for each state s. Thus we cannot simply apply Algorithm 1 to the weak transition relation.

Of course it is easy to see that these τ self-loops will be the only cycles. A naive approach would then simply remove these self-loops from the weak transition relation and apply Algorithm 1 to this modified weak transition relation. This approach however does not work. Consider IMC $\mathcal{P}$ in Figure 1. It is obvious that states s_1 and s_3 are weakly bisimilar, while the naive approach would decide that they are not, since s_1 can do the weak transition $s_1 \stackrel{\tau}{\Longrightarrow} s_3$, which s_3 seemingly cannot simulate if we do not consider the τ-loop $s_3 \stackrel{\tau}{\Longrightarrow} s_3$. Even if we would memorize that each state has a τ-loop, and treat this case separately in the algorithm, this is not enough, since there is a fundamental difference to the strong case. We find in fact that Theorem 1 does not hold for weak bisimulation! States s_1 and s_3 have different ranks (2 and 1 respectively) but they are still weakly bisimilar. We can however, define a different ranking of states for which a similar theorem does hold.

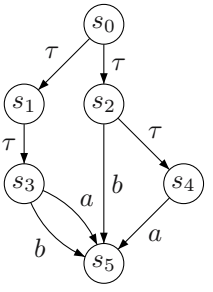


Fig. 1. An acyclic IMC $\mathcal{P}$

4.1 Weak Ranks

Let $\mathcal{P} = \langle S, s^0, A, R_i, R_M \rangle$ be an acyclic IMC. To find the *weak rank* of a state we do not look at the longest path starting in that state but we find the longest path counting only the observable transitions.

Definition 6. *We define the notion of* weak rank *(or the observable rank)* $\mathsf{R}^W : S \to \mathbb{N}$ *of a state* s *as the maximum weak length of all plausible paths starting in* s*:* $\mathsf{R}^W(s) = max\{wlen(\pi) \mid \pi \in Paths^P(s)\}$.

For weak ranks we can establish a theorem similar to Theorem 1.

Theorem 3. *If two states of* $\mathcal{P}$ *are weakly bisimilar they have the same weak rank:* $\forall s, t \in S \cdot s \approx t \to \mathsf{R}^W(s) = \mathsf{R}^W(t)$.

For strong bisimulation and ranks we found the property that strong bisimulation for states on a certain rank only depends on the states below this rank. Unfortunately, this property does not hold for weak ranks. Consider again IMC $\mathcal{P}$ in Figure 1. State s_5 has weak rank 0 whereas all other states have weak rank 1. If we consider only weak transitions to the states on rank 0 we are tempted to conclude that s_0, s_1, s_2 and s_3 are all weakly bisimilar, since they can all do the weak moves $\stackrel{a}{\Longrightarrow} s_5$ and $\stackrel{b}{\Longrightarrow} s_5$, while state s_4 can only do the weak move $\stackrel{a}{\Longrightarrow} s_5$. However state s_0 can also do the move $s_0 \stackrel{\tau}{\Longrightarrow} s_4$ which s_1, for instance, cannot simulate and thus s_0 and s_1 are actually not weakly bisimilar.

4.2 A Different Way of Partitioning the State Space

To make use of the acyclicity of our models in computing weak bisimulation we need to order the state space such that *(i)* weak bisimilarity of states on order x only depends on the bisimilarity of states of a lower order, and *(ii)* all states that are weakly bisimilar have the same order. However, we have seen that the weak rank does not satisfy the first requirement and the rank does not satisfy the second.

We introduce the notion of *level* of a state, which is computed on-the-fly, to order the state space. A level function L maps states to some ordered well-founded set W, such that for state s, level $L(s)$ is the smallest value satisfying the following conditions

1. State s has no outgoing transitions to any other state t on a level higher than $L(s)$: $s \xrightarrow{a} t$ implies $L(s) \geq L(t)$.
2. State s has no outgoing observable transitions to any state t on *level* $L(s)$: $s \xrightarrow{a} t \wedge L(s) = L(t)$ implies $a = \tau$.
3. State s has no outgoing τ-transitions to any state t on *level* $L(s)$, unless s is weakly bisimilar to t: $s \xrightarrow{\tau} t \wedge L(s) = L(t)$ implies $s \approx t$.

Notably, partitioning the state space in such levels satisfies the two requirements *(i)* and *(ii)* above: If a state s has a level higher than t they cannot be weakly bisimilar since s must have a transition to a non-bisimilar state that is at least on the same level as t. Furthermore the bisimilarity of two states on a level i depends only on lower levels since weak transitions to bisimilar states can, by definition, always be simulated. However, if we want to use such a level function to compute the weak bisimulation for an acyclic IMC $\mathcal{P} = \langle S, s^0, A, R_i, R_M \rangle$, we are in a trap, because condition 3 relies on knowledge about the weak bisimulation classes.

Our algorithm exploits that we can – for a particular level – obtain the required bisimulation knowledge by only looking at an IMC that is spanned by lower levels. This allows us to increment our knowledge about both L and $\approx$ while 'climbing up' the transitions, starting with absorbing states. Technically, we are working with a sequence of partial functions $L'_1, \ldots, L'_k$ $(L'_i : S \to W)$ that satisfy (in set notation) $L'_i \subset L'_{i+1}$ and that must converge towards the total function L in finitely many steps. For a given partial function L', and a fixed level $w \in W$, the IMC spanned by level w is defined by restricting the transitions of $\mathcal{P}$ to only those transitions which are part of the weak transition relation to states s with level $L'(s) \leq w$.

Definition 7. *For an acyclic IMC $\mathcal{P}$, a partial function $L' : S \to W$ to an ordered well-founded set, and an element of the ordered set $w \in W$ the* IMC spanned by level w *is $\mathcal{P}_w = \langle S, s^0, A, R_i^w, R_M^w \rangle$, where:*

$$
\begin{aligned}
R_i^w &= \{(s,\tau,t) \mid \exists t' \cdot t \stackrel{x}{\Longrightarrow} t' \wedge L'(t') \leq w \wedge (s,\tau,t) \in R_i\} \\
&\quad \cup \{(s,a,t) \mid \exists t' \cdot t \stackrel{\tau}{\Longrightarrow}{}^{i} t' \wedge L'(t') \leq w \wedge (s,a,t) \in R_i\} \\
R_M^w &= \{(s,\lambda,t) \mid \exists t' \cdot t \stackrel{\tau}{\Longrightarrow}{}^{i} t' \wedge L'(t') \leq w \wedge (s,\lambda,t) \in R_M\}
\end{aligned}
$$

Algorithm 2. Determining the weak bisimilarity quotient for an acyclic IMC

Require: $\mathcal{P} = \langle S, s^0, A, R_i, R_M \rangle$ is an acyclic IMC.

1: $(\mathsf{R}^W, \#wout) = \textsc{ComputeRanks}(\mathcal{P})$
2: $maxwrank = max\{n \mid s \in S \wedge \mathsf{R}^W(s) = n\}$
3: $BLOCKS = \langle\{\{s \mid s \in S \wedge \mathsf{R}^W(s) = n\}\} \mid n \leftarrow \langle 0 \ldots maxwrank \rangle\rangle$
4: $matrix = \underline{0}$; $rate = \underline{0}$
5: **for** $i = 0$ to $maxrank$ **do**
6: $\quad LSTATES = \{s \mid \mathsf{R}^W(s) = i \wedge \#wout(s) = 0\}$
7: $\quad NSTATES = \emptyset$
8: $\quad j = 0$
9: $\quad$ **while** $LSTATES \neq \emptyset$ **do**
10: $\quad\quad \textsc{ComputeLevel}(\mathcal{P}, BLOCKS[i], LSTATES, NSTATES)$
11: $\quad\quad LBLOCKS = \{\{s \mid s \in B \wedge s \in LSTATES\} \mid B \in BLOCKS[i]\}$
12: $\quad\quad$ **for all** $(s, a, B) \in \{(s, a, B) \mid B \in LBLOCKS \wedge \exists t \in B \cdot s \stackrel{a}{\Longrightarrow}^i t\}$ **do**
13: $\quad\quad\quad matrix[s][a][B] = 1$
14: $\quad\quad$ **for all** $(s, \lambda, B) \in \{(s, \lambda, B) \mid B \in LBLOCKS \wedge \exists t \in S \cdot s \stackrel{\tau}{\Longrightarrow}^i t \wedge t$ stable $\wedge$ $\gamma_M(t, B^\tau) = \lambda\}$ **do**
15: $\quad\quad\quad rate[s][B] = \lambda$
16: $\quad\quad$ **for** $k = i$ to $maxrank$ **do**
17: $\quad\quad\quad BLOCKS[k] = \bigcup\{\{\{t \mid t \in B \wedge matrix[t] = matrix[s] \wedge rate[t] = rate[s]\} \mid s \in B\} \mid B \in BLOCKS[k]\}$
18: $\quad\quad LSTATES = \{s \mid s \in NSTATES \wedge \#wout(s) = 0\}$
19: $\quad\quad NSTATES = \emptyset$
20: $\quad\quad j = j + 1$

Our intention is to reformulate conditions 2 and 3 above to the following: if a state s has a level $L'(s) = w$ and w' is the largest value smaller than w appearing in the range of L', then for all transitions $s \xrightarrow{\tau} t$ we find:

1. State t has a level strictly lower than w, or
2. State t has level w and is weakly bisimilar to s on the IMC spanned by level w'.

This property holds indeed for our algorithm, because the sequence $L'_1, \ldots, L'_k$ is constructed level-by-level, starting from the bottom level. The algorithm also implies that if $L'(s)$ is defined, then L' is defined for all states reachable from s as well, which is a requirement for the above idea to work.

Algorithm 2 computes the levels while traversing the state space, and while computing the weak bisimulation relation for an acyclic IMC. Line 1 calculates the weak rank and the number of outgoing weak transitions for every state. Notably, the levels (line 10) and the weak bisimulation relation (lines 12 to 17) are calculated per weak rank. This is justified by Theorem 3. In other words, we fix $W = \mathbb{N} \times \mathbb{N}$ where the first component is the weak rank, and use the (lexicographic) order on pairs $(x, y) \geq (x', y') \iff x > x' \vee (x = x' \wedge y \geq y')$. For each iteration of the loop 9–20 the set $LSTATES$ then contains exactly those states which have level (i, j), see Definition 8 below, and the set $LBLOCKS$ partitions $LSTATES$ into sets of weakly bisimilar states. The set $NSTATES$

Algorithm 3. ComputeLevel($\mathcal{P}, BLOCKS, LSTATES, NSTATES$)

1: **for all** States s in $LSTATES$ **do**
2: **for all** Transitions $t \xrightarrow{\tau} s \in R_i$ with $\mathsf{R}^W(t) = \mathsf{R}^W(s)$ **do**
3: $\#wout(t) = \#wout(t) - 1$
4: **if** $\neg\exists B \in BLOCKS \cdot s, t \in B$ **then**
5: $NSTATES = NSTATES \cup \{t\}$
6: **else**
7: **if** $\#wout(t) = 0 \wedge t \notin NSTATES$ **then**
8: $LSTATES = LSTATES \cup \{t\}$

contains all states with weak rank i, level greater than (i, j) and at least one transition to a state on level (i, j).

Theorem 4. *Given an acyclic IMC $\mathcal{P}$, Algorithm 2 computes the weak bisimulation correctly. Moreover, the time-complexity of the algorithm, given the weak transition relation, is $\mathcal{O}(n^2)$. The space complexity is $\mathcal{O}(n^2)$.*

4.3 Correctness

We give here an extended sketch of the proof of correctness for Algorithm 2. For the full proof we refer to [8]. First we define the notion of the level of a state based on the two conditions given at the end of Subsection 4.2.

Definition 8 (Level (i, j)). *Let $\mathcal{P} = \langle S, s^0, A, R_i, R_M \rangle$ be an acyclic IMC. We define the set of all states in $\mathcal{P}$ with* level (i, j)*, written $\mathcal{L}_{(i,j)}$ as the largest set for which the following holds:*

$$s \in \mathcal{L}_{(i,j)} \rightarrow \mathsf{R}^W(s) = i \wedge \neg\exists j' < j \cdot s \in \mathcal{L}_{(i,j')} \wedge \forall s \xrightarrow{x} t \cdot L(t) < (i, j) \vee (t \in \mathcal{L}_{(i,j)} \wedge s \approx t)$$

We write $L(s) = (i, j)$ if and only if $s \in \mathcal{L}_{(i,j)}$.

To prove that Algorithm 2 is correct we prove that it computes in each iteration (i, j) of the loop 9–20, the set of states on level (i, j) $LSTATES$ (line 10) and weak bisimulation on the IMC spanned by level (i, j) $BLOCKS$ (lines 16–17). By computing the set of states on level (i, j) we also further refine the partial function L' in each iteration, which is initially completely undefined. By iteration (i, j) we mean that pass of loop 9–20 where variables i and j have those particular values. Line 1 computes, for each state, its weak rank and the number of outgoing transitions to states on the same weak rank ($\#wout(s)$). For each weak rank i the loop 9–20 terminates having computed the weak bisimulation on the IMC spanned by the maximum possible level for weak rank i, denoted (i, max_i). The algorithm then terminates after computing weak bisimulation for the IMC spanned by the maximal level for the maximal weak rank which is equivalent to the IMC itself.

First we consider the computation of weak bisimulation on IMCs spanned by the different levels. Line 3 initializes $BLOCKS$ such that all states are partitioned

according to weak rank, this is justified by Theorem 3. For level $(0,0)$ weak bisimulation on $\mathcal{P}_{(0,0)}$ is computed in lines 12–19. For a level $(i,0) > (0,0)$ we compute weak bisimulation on $\mathcal{P}_{(i,0)}$ by refining weak bisimulation on $\mathcal{P}_{(i-1,\mathsf{max}_{i-1})}$. We do this by considering all the weak transitions to states on level $(i,0)$ in lines 12–15[1], note that we compute the set of states on level $(i,0)$ in that same iteration on line 10. Here it is assumed that the partition of states on level $(i,0)$, *LBLOCKS*, is the partition according to weak bisimilarity on $\mathcal{P}$. This is correct since the only outgoing weak transitions for states on level $(i,0)$ we have not yet considered are those that go to other states on level $(i,0)$. But, by Definition 8, such transitions are τ transitions which go to bisimilar states and such transitions can always be simulated. This then means that in line 17 weak bisimulation on $\mathcal{P}_{(i,0)}$ is computed. The same holds for iteration (i,j) with $j > 0$ where weak bisimulation on $\mathcal{P}_{(i,j-1)}$ is refined to weak bisimulation on $\mathcal{P}_{(i,j)}$.

Now we consider the computation of levels. For every weak rank i line 6 initializes *LSTATES* to all states on weak rank i with only transitions to states levels lower than $(i,0)$. By definition these states must have level $(i,0)$. Line 7 initializes *NSTATES* to $\emptyset$. The function ComputeLevel then considers all τ-transitions to states in *LSTATES*. If a state is found which has a τ-transition to a non-bisimilar state - with respect to transitions to states on levels below $(i,0)$, and note that we have computed this relation in the previous iteration - on level $(i,0)$ then we add this state to *NSTATES* since we are sure it has level higher than $(i,0)$. If the condition of line 7 of ComputeLevel is met for a state t then all outgoing transitions of t must go to lower levels or to bisimilar states on level $(i,0)$ which means that t also has level $(i,0)$. When ComputeLevel terminates we must have found all states on level $(i,0)$ because the model is acyclic. Now *NSTATES* contains all states with at least one transition to level $(i,0)$. For those states s in *NSTATES* which now have $\#wout(s) = 0$ (line 18 of Algorithm 2) we know that they only have transitions to states lower than $(i,1)$ and thus they must have level $(i,1)$. In this way we compute all the levels recursively. For each weak rank loop 9–20 must terminate since there are only a finite number of states on that weak rank. Acyclicity also ensures that we must encounter and assign a level to all states in the function ComputeLevel.

This shows that in each iteration (i,j) of loop 9–20 the states on level (i,j) and $\mathcal{P}_{(i,j)}$ are computed. Finally then weak bisimulation on $\mathcal{P}$ spanned by the maximum level (i_m, j_m) is computed. Since all states must have a level smaller than or equal to the maximum we find that $\mathcal{P}_{(i_m,j_m)} = \mathcal{P}$.

4.4 Complexity

We first discuss the complexity of the algorithm itself. Afterwards we discuss the time needed to compute the weak transition relation. In the following n, m and l are used the denote the number of states ($|S|$), transitions ($|R_i| + |R_M|$) and the number of actions ($|A|$) respectively.

[1] We only consider the weak transition relation to states on level $(i,0)$ while $\mathcal{P}_{(i,0)}$ also contains transitions above this level. However, we will consider these transitions in later iterations of the algorithm.

The ranks are computed with a simple depth-first search which can be done in time $\mathcal{O}(m)$. The level computations are also done in time $\mathcal{O}(m)$ as each state is evaluated exactly once in Algorithm 3 and in each such evaluation all incoming transitions are considered once. For the partitioning of the state space in weakly bisimilar blocks every state is considered exactly once, since we only consider states in blocks that we do not have to partition anymore. For each state we must consider each incoming weak move in the weak transition relation. The time complexity is then in the order of the size of the weak transition relation which is $\mathcal{O}(n^2)$. There can be at most n partitions (in the case that there are no bisimilar states at all) so we must make at most n partitions which then takes $\mathcal{O}(n)$ time.

There are at most $\mathcal{O}(ln^2)$ transitions in an acyclic IMC. If we consider the number of actions to be constant then m will be in $\mathcal{O}(n^2)$ which proves that Algorithm 2 computes the weak bisimulation quotient for an acyclic IMC in time $\mathcal{O}(n^2)$ given that we know the weak transition relation. For the space complexity we find that we need to store several attributes of the states, but most importantly we must store the weak transition relation which is again of size $O(n^2)$.

However computing the weak transition relation theoretically dominates the rest of the algorithm in the general case as well as the acyclic case. We will see in Section 6 that this is usually not the case in practice, as also noted in [15]. In the general case the best theoretical complexity for computing the reflexive transitive closure is $\mathcal{O}(n^{2.376})$ as given by [6]. For acyclic graphs [23] gives an algorithm which computes the reflexive transitive closure in time $\mathcal{O}(mk)$ where k is the size of the chain decomposition of the model (which is at most n). In our implementation we have adapted the simple algorithm in [13] which has worst-case time complexity $\mathcal{O}(mn)$. Theoretically the algorithm is then still cubic in the number of states, but we will discuss why this is often not the case in practice.

For acyclic models m^*, the number of transitions in the weak transition relation is at most $\mathcal{O}(ln^2)$. For some state s, m_s^*, the number of outgoing weak transitions of s will be at most nl. As in [13] we compute $\Longrightarrow$ by starting at the states with rank 0 and then moving up the ranks. This means that for any state s we will compute $out^*(s)$ (the outgoing weak moves of state s) by merging $out^*(t')$ for all transitions $s \rightarrow t'$. This is possible since the state t' must have a lower rank than s. We will then find every move $s \overset{a}{\Longrightarrow} t$ at most once for each outgoing transition. This means that the complexity is $\mathcal{O}(nm^*) = \mathcal{O}(n^3)$. However, since we have different action labels, the situation is more nuanced. We can in fact only find the move $s \overset{a}{\Longrightarrow} t$ for outgoing τ- or a-transitions. So we will find every move $s \overset{a}{\Longrightarrow} t$ exactly $m_{s,\tau} + m_{s,a}$ times, where $m_{s,\tau}$ is the number of outgoing τ-transitions of state s and $m_{s,a}$ the number of outgoing a-transitions. For a state s and an action a we can find the, at most n, a-moves in m_s^* at most $m_{s,\tau} + m_{s,a}$ times. If we now sum over all actions and all states we find:

$$\sum_{a}^{a \in A \setminus \{\tau\}} \sum_{s}^{s \in S} nm_{s,\tau} + nm_{s,a} = \sum_{a}^{a \in A \setminus \{\tau\}} nm_\tau + nm_a = lnm_\tau + nm_{A \setminus \{\tau\}}$$

In the worst case $lnm_\tau + nm_{A\setminus\{\tau\}}$ is still cubic in the number of states. However, we can make an interesting observation about models which are observably deterministic, i.e. each state has at most one outgoing transition for each visible action. We then find that $m_{A\setminus\{\tau\}}$ is at most ln and, if we then assume the number of actions is constant we find complexity $\mathcal{O}(nm_\tau + n^2)$. The models used in the dynamic fault tree domain [4] are all observably deterministic.

5 Extensions to the Algorithm

The algorithm presented in this paper was developed with a particular application in mind, namely dynamic fault tree analysis. We have therefore extended the algorithm to compute the desired equivalence relation, weak Markovian bisimulation, for the desired formalism, IOIMCs.

Weak Markovian bisimulation. Weak Markovian bisimulation for IMCs (introduced as weak bisimulation in [5], refereed to here as weak Markovian bisimulation to avoid confusion) differs from weak bisimulation in that Markovian transitions to otherwise bisimilar states are not observable. This bisimulation relation is justified by the fact that IMCs are memoryless in time [16]. A Markovian transition from state s to state t means that we may move from s to t after some time. If we find, however, that the behavior of s and t are the same then this extra time span does not influence the behavior, since the memoryless property ensures that the behavior of t does not depend on the time we arrive in state t.

Adapting our algorithm to weak Markovian bisimulation is very simple. All that needs to be changed is that in the adapted algorithm Markovian transitions are considered unobservable, like τ transitions. This only has a direct effect on the definition, and computation, of weak ranks (see Definition 6).

Input/output interactive Markov chains. In compositional dynamic fault tree (DFT) analysis [4] the components of a DFT are modelled using IOIMCs. The IOIMC formalism is a variant of the IMC formalism, inspired by I/O automata [19], in which the actions of a model are divided into input, output and internal transitions. The difference between weak Markovian bisimulation for IMCs and for IOIMCs is that for IOIMCs there may be more internal actions, while IMCs only use τ and the maximal progress assumption is also applied to output transitions. The first difference is handled by renaming all the different internal actions of an IOIMC to τ. The second difference is covered by refining the stability check in Algorithm 2 (line 17).

Of course we can apply our algorithm in DFT analysis only if the models encountered are acyclic. At first glance this is the case indeed, since a DFT describes the degradation of a system towards system failure, i.e. how the interplay of component failures may lead to a (possibly catastrophic) system failure. Since repairs are not considered, the behaviour is structurally acyclic.

There is a fine point because input-enabledness (typical of I/O automata) leads to models that may choose to do nothing in response to certain inputs.

This means that the state does not change, so the models are actually not acyclic. However, using the fact that the models are input-enabled and observably deterministic (see [4]), allows us to cut input loops without losing any information. To deal with the fact that two states $s \xrightarrow{a?} t$ may be bisimilar we make input-actions unobservable (just as we did for Markovian transitions). The adapted algorithm also maintains the information on removed input self-loops in order to properly build the weak transition relation.

6 Experimental Results

In this section we show, with a number of case studies that for acyclic models our algorithm is much more efficient than the existing algorithms. We compare the performance of the acyclic algorithm in minimising IOIMCs in the context of DFT analysis with the BCG_MIN tool from the CADP toolset [12]. We have so far been using BCG_MIN in the analysis of DFTs within the CORAL tool [4], which computes branching bisimulation for (IO)IMCs. However a weak bisimulation minimiser fits the theory better. Branching bisimulation is usually faster to compute than weak bisimulation and will give the same numerical results for DFT analysis, although intermediate models may be larger when using branching bisimulation. All experiments were run on a Linux machine with an AMD Athlon(tm) XP 2600+ processor at 2 GHz equipped with 2GB of RAM. In the following tables, n, m and m' stand for the number of states, transitions and the size of the weak transition relation from states in the input model to states in the reduced output model. m' is a lower bound on the size m^* of the weak transition relation of the input model. The computation of the latter is avoided by our algorithm, so we cannot report the sizes.

The first example we consider is a large version of a fault-tolerant parallel processor case study (FTPP-6) from [4]. It has 6 network elements and 6 groups of 4 processors. As explained earlier, the state spaces are constructed by compositional aggregation, where composition and minimisation steps are intertwined. Using the BCG_MIN tool in this process, leads to an overall time consumption of 7 hours, 3 minutes and 48 seconds. Using our acyclic minimization algorithm instead the same process only required 30 minutes and 20 seconds. Some insight in this 14-fold speed up is provided in the first rows of Table 1, where some representative intermediate state spaces and their reduction times are listed.

Another case study, an extended version of the cardiac assist system case study from [4] (CAS-l) is considered below. We found that the overall time for compositional aggregation decreased from 1 hour 57 minutes, 13 seconds (using BCG_MIN) to 1 minute 53 seconds using our dedicated algorithm. Again some intermediate steps are listed in the table.

For a larger version of the multi-processor distributed computing system case study (MDCS-l), also taken from [4], we found that the acyclic algorithm requires 5 minutes, 41 seconds to do compositional aggregation while BCG_MIN requires 30 minutes and 9 seconds. Table 1 shows the timing results for the minimization of various intermediate models.

Table 1. Minimisation times for acyclic IOIMCs in seconds

Case study	n	m	m'	BCG_MIN	acyclic
FTPP-6	9237	68419	93214	18.21	0.67
	32909	293955	398661	129.09	2.34
	79111	1324359	1364850	104.43	9.36
	101426	1043520	1402507	630.33	12.52
	255397	2920996	3883860	2028.29	83.60
	464762	5756020	7553746	4696.40	289.36
	464762	6066486	7833720	3742.70	222.15
	1180565	22147378	22502816	13631.44	666.28
CAS-1	38396	389714	482657	160.22	5.73
	61055	378219	548331	4455.71	6.27
	62860	419459	601135	1457.25	6.94
	66137	483157	672742	688.83	8.05
	57373	675330	1310517	14.05	5.42
MDCS-1	26466	244003	326916	13.37	1.39
	55578	645119	856984	41.78	3.3
	99242	1395459	1816748	75.21	6.98
	97482	1606231	2049696	101.38	9.38

As evident from the table and the reported time savings, the effectiveness of bisimulation minimization and of compositional aggregation is improved drastically for acyclic models.

7 Conclusion

This paper has developed bisimulation minimisation algorithms for acyclic IMC models. While this work is motivated in a very concrete application, namely the analysis of very large dynamic fault tree specifications, the results equally apply to acyclic labelled transition systems. We are rather happy with the speedup achieved over algorithms for possibly cyclic models. One interesting question we have not yet considered is how the algorithm can be twisted towards branching bisimulation, possibly also exploring links to normed simulation [14] and the cones-and-foci method [11]. Another promising avenue of research lies in extending the algorithm to also minimise cyclic models along the same lines as [9].

References

1. Blom, S., Fokkink, W., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: μcrl: A toolset for analysing algebraic specifications. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 250–254. Springer, Heidelberg (2001)
2. Blom, S., Orzan, S.: Distributed branching bisimulation reduction of state spaces. Electr. Notes Theor. Comput. Sci. 89(1) (2003)
3. Böde, E., Herbstritt, M., Hermanns, H., Johr, S., Peikenkamp, T., Pulungan, R., Wimmer, R., Becker, B.: Compositional performability evaluation for statemate. In: QEST, pp. 167–178 (2006)
4. Boudali, H., Crouzen, P., Stoelinga, M.: A compositional semantics for dynamic fault trees in input/output interactive markov chains. In: ATVA, pp. 441–456 (2007)
5. Bravetti, M.: Revisiting interactive markov chains. Electronic Notes in Theoretical Computer Science 68(5) (2002)
6. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: ACM Symposium on Theory of Computing (1987)

7. Coste, N., Garavel, H., Hermanns, H., Hersemeule, R., Thonnart, Y., Zidouni, M.: Quantitative evaluation in embedded system design: Validation of multiprocessor multithreaded architectures. In: DATE (2008)
8. Crouzen, P., Hermanns, H., Zhang, L.: No Cycling? Go Faster! On the minimization of acyclic models. Reports of SFB/TR 14 AVACS 41, SFB/TR 14 AVACS (July 2008), http://www.avacs.org ISSN: 1860-9821
9. Dovier, A., Piazza, C., Policriti, A.: A fast bisimulation algorithm. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 79–90. Springer, Heidelberg (2001)
10. Dugan, J., Bavuso, S., Boyd, M.: Dynamic fault-tree models for fault-tolerant computer systems. IEEE Transactions on Reliability 41(3), 363–377 (1992)
11. Fokkink, W., Pang, J., van de Pol, J.: Cones and foci: A mechanical framework for protocol verification. Formal Methods in System Design 29(1), 1–31 (2006)
12. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: Cadp 2006: A toolbox for the construction and analysis of distributed processes. In: CAV (2006)
13. Goralcikova, A., Koubek, V.: A reduct-and-closure algorithm for graphs. Mathematical Foundations of Computer Science 74, 301–307 (1979)
14. Griffioen, W.O.D., Vaandrager, F.W.: A theory of normed simulations. ACM Trans. Comput. Log. 5(4), 577–610 (2004)
15. Groote, J.F., Vaandrager, F.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443. Springer, Heidelberg (1990)
16. Hermanns, H.: Interactive Markov Chains: The Quest for Quantified Quality. LNCS, vol. 2428. Springer, Heidelberg (2002)
17. Katoen, J.-P., Kemna, T., Zapreev, I.S., Jansen, D.N.: Bisimulation minimisation mostly speeds up probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007)
18. Lang, F.: Refined interfaces for compositional verification. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 159–174. Springer, Heidelberg (2006)
19. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
20. Mateescu, R.: Local model-checking of modal mu-calculus on acyclic labeled transition systems. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 281–295. Springer, Heidelberg (2002)
21. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
22. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
23. Simon, K.: An improved algorithm for transitive closure on acyclic digraphs. In: Kott, L. (ed.) ICALP 1986. LNCS, vol. 226, pp. 376–386. Springer, Heidelberg (1986)
24. Vesely, W., Goldberg, F., Roberts, N., Haasl, D.: Fault Tree Handbook. NASA (1981)
25. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref- a symbolic bisimulation tool box. In: ATVA, pp. 477–492 (2006)

Quasi-Static Scheduling of Communicating Tasks

Philippe Darondeau[1], Blaise Genest[1], P.S. Thiagarajan[2], and Shaofa Yang[1]

[1] IRISA, CNRS & INRIA, Rennes, France
[2] School of Computing, National University of Singapore

Abstract. Good scheduling policies for distributed embedded applications are required for meeting hard real time constraints and for optimizing the use of computational resources. We study the *quasi-static scheduling* problem in which (uncontrollable) control flow branchings can influence scheduling decisions at run time. Our abstracted task model consists of a network of sequential processes that communicate via point-to-point buffers. In each round, the task gets activated by a request from the environment. When the task has finished computing the required responses, it reaches a pre-determined configuration and is ready to receive a new request from the environment. For such systems, we prove that determining existence of quasi-static scheduling policies is undecidable. However, we show that the problem is decidable for the important sub-class of "data branching" systems in which control flow branchings are due exclusively to data-dependent internal choices made by the sequential components. This decidability result—which is non-trivial to establish—exploits ideas derived from the Karp and Miller coverability tree [8] as well as the existential boundedness notion of languages of message sequence charts [6].

1 Introduction

We consider systems that consist of a finite collection of processes communicating via point-to-point buffers. Each process is a sequential transition system, in which non-deterministic branchings may be of two types: (i) a data-dependent internal choice made by a sequential component; (ii) a process waiting for messages on different input buffers. In the second case, the waiting process non-deterministically branches by picking up a message from one of the *nonempty* input buffers [3]. The system of processes is triggered by an environment iteratively in *rounds*. We model the system dynamics for just one round. It is easy to lift our results to multiple rounds. In each round, the environment sends a data item to one of the processes. This starts the computations done in the round. When the computation finishes, all the processes are in their final states and the buffers are empty. In a technical sense, buffers—viewed as counters without zero tests—are deployed here as over-approximated abstractions of FIFOs. We note that using FIFOs or zero tests would render the model Turing powerful [1].

In this setting, one is interested in determining a schedule for the processes. If at a configuration the scheduler picks the process p to execute and p is at a

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 310–324, 2008.

state with several outgoing transitions, then the schedule must allow *all* possible choices to occur. As a result, such schedules are referred to as *quasi-static* schedules. In addition, the schedule should never prevent the system from (eventually) reaching the final state. We deem such schedules to be *valid*. In addition, a quasi-static schedule is required to be *regular* in the sense that the system under schedule should use only a *bounded* amount of memory to service the request from the environment. In particular, the schedule should enforce a uniform bound on the number of items stored in the buffers during the round.

Our first result is that determining whether a valid and regular quasi-static schedule exists is undecidable. In fact the undecidability result holds even if the system by itself is valid in that from every reachable global state it is possible to reach the final global state; the schedule does not need to enforce this. Next we define data-branching systems in which the only branching allowed is local (data) branching; simultaneous polling on multiple input buffers is ruled out. We show that for data-branching systems, one can effectively check whether there exists a valid and regular quasi-static schedule. This result is obtained using classical ideas from [8] and by exploiting a special scheduling policy, called the canonical schedule. The canonical schedule is very similar to a normal form obtained for determining the existential boundedness property of certain languages of message sequence charts [6]. The crucial point here is that one cannot directly apply the techniques of [8] because the canonical schedule uses zero tests on buffers. Whereas, as is well known, it is often the case that zero tests lead to undecidability.

Before considering related work, it is worth noting that our setting is strongly oriented towards distributed tasks and their rounds-based executions. Hence it does not cater for models capturing non-terminating computations such as Kahn process networks [7]. At present, it is not clear whether our undecidability result can be extended to such settings. Quasi-static scheduling (QSS) has been studied before in a number of settings (see [9] for a survey). An early work in [2] studied dynamic scheduling of boolean-controlled dataflow (BDF) graphs. Being Turing powerful, the QSS problem for this class of systems is undecidable [2]. Later, [3] proposed a heuristic to solve QSS on a different model called the YAPI model by exploring only a subset of the infinite state space. There is however no proof that the heuristic is complete even on a subset of YAPI models. The work [10] considered QSS on a restricted class of Petri nets called Equal-Conflict Petri nets and showed decidability. However the notion of schedulability used in [10] is much weaker than the one in [3] or ours. Basically, under the scheduling regime defined in [10], only a finite number of runs can arise, hence in effect, systems with loops are not schedulable. In comparison, our system model is very close to (general) Petri Nets. Our scheduling notion is essentially the one presented in [3], slightly modified to fit our model. Our undecidability result is also harder to obtain than the one in [2], since reachability is decidable for our model. Indeed, the decidability of this quasi-static schedulability problem is stated as an open problem in [3, 9]. The work [12] considered QSS with the setting of [3] and

proposed a sufficient (but not necessary) condition for non-schedulability based on the structure of the Petri net system model.

In the next section we present our model and the quasi-static scheduling problem. Section 3 establishes the undecidability result in the general setting. Section 4 imposes the data-branching restriction and shows the decidability of the quasi-static scheduling problem under this restriction. The final section summarizes and discusses our results. Proofs omitted, due to lack of space, can be found in [4].

2 Preliminaries

Through the rest of the paper, we fix a finite set $\mathcal{P}$ of process names. Accordingly, we fix a finite set Ch of buffer names. To each buffer c, we associate a source process and a destination process, denoted $src(c)$ and $dst(c)$ respectively. We have $src(c) \neq dst(c)$ for each $c \in Ch$. For each p, we set $\Sigma_p^! = \{!c \mid c \in Ch, src(c) = p\}$ and $\Sigma_p^? = \{?c \mid c \in Ch, dst(c) = p\}$. So, $!c$ stands for the action that deposits one item into the buffer c while $?c$ is the action that removes one item from c. For each p, we fix also a finite set Σ_p^{cho} of choice actions. We assume that $\Sigma_p^{cho} \cap \Sigma_q^{cho} = \emptyset$ whenever $p \neq q$. Members of Σ_p^{cho} will be used to label branches arising from abstraction of "if...then...else", "switch..." and "while..." statements. For each p, we set $\Sigma_p = \Sigma_p^! \cup \Sigma_p^? \cup \Sigma_p^{cho}$. Note that $\Sigma_p \cap \Sigma_q = \emptyset$ whenever $p \neq q$. Finally, we fix $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$.

A *task system* (abbreviated as "system" from now on) is a structure $\mathcal{A} = \{(S_p, s_p^{in}, \longrightarrow_p, s_p^{fi})\}_{p \in \mathcal{P}}$, where for each $p \in \mathcal{P}$, S_p is a finite set of states, s_p^{in} is the initial state, $\longrightarrow_p \subseteq S_p \times \Sigma_p \times S_p$ is the transition relation, and s_p^{fi} is the final state. As usual, if $s_p \in S_p$ and $\delta = (\hat{s}_p, a_p, \hat{s}'_p)$ is in $\longrightarrow_p$ with $\hat{s}_p = s_p$, then we call δ an outgoing transition of s_p. We require the following conditions to be satisfied:

- For each $p \in \mathcal{P}$ and $s_p \in S_p$, if the set of outgoing transitions of s_p is not empty, then exactly one of the following conditions holds:
  - Each outgoing transition of s_p is in $S_p \times \Sigma^{cho} \times S_p$. Call such an s_p a *(data-dependent) choice* state.
  - s_p has precisely one outgoing transition $(s_p, !c, s'_p)$, where $c \in Ch, s'_p \in S_p$. Such an s_p is called a *sending* state.
  - Each outgoing transition of s_p is in $S_p \times \Sigma_p^? \times S_p$. Call such an s_p a *polling* state.
- For each process p, its final state s_p^{fi} either has no outgoing transitions or is a polling state.

Intuitively, the system works in rounds. A round starts as if a message from the environment had just been received. At its final state, a process p should stay watching buffers for messages possibly sent by other processes. If every process is in its final state, and all buffers are empty, a reset operation triggered by the environment may be performed to start a new round. This operation puts

every process in its initial state from which the computation can start again. Thus, computations belonging to different rounds will not get mixed up. (We do not explicitly represent this reset operation in the system model.) For technical convenience, we do not consider multi-rate communications, that is, multiple items can be deposited to or picked up from a buffer at one time. However, our results extend to multi-rate task systems easily.

For notational convenience, we shall assume that the system is *deterministic*, that is for each p, for each $s_p \in S_p$, if $(s_p, a1, s1_p)$, $(s_p, a2, s2_p)$ are in $\longrightarrow_p$, then $a1 = a2$ implies $s1_p = s2_p$. All our results can be extended easily to non-deterministic systems. The dynamics of a system $\mathcal{A}$ is defined by the transition system $TS_{\mathcal{A}}$. A configuration is (s, χ) where $s \in \prod_{p\in\mathcal{P}} S_p$ and χ is a mapping assigning a non-negative integer to each buffer in Ch. We term members of $\prod_{p\in\mathcal{P}} S_p$ as *global* states. We view a member s of $\prod_{p\in\mathcal{P}} S_p$ as a mapping from $\mathcal{P}$ to $\bigcup_{p\in\mathcal{P}} S_p$ such that $s(p) \in S_p$ for each p. When no confusion arises, we write s_p for $s(p)$. The *initial* configuration is (s^{res}, χ^0) where $s^{res}(p) = s_p^{res}$ for each p. Further, $\chi^0(c) = 0$ for every $c \in Ch$. We define $TS_{\mathcal{A}} = (RC_{\mathcal{A}}, (s^{res}, \chi^0), \Longrightarrow_{\mathcal{A}})$ where the (possibly infinite) set $RC_{\mathcal{A}}$ of reachable configurations and $\Longrightarrow_{\mathcal{A}} \subseteq RC_{\mathcal{A}} \times \Sigma \times RC_{\mathcal{A}}$ are the least sets satisfying the following:

- $(s^{res}, \chi^0) \in RC_{\mathcal{A}}$.
- Suppose configuration (s, χ) is in $RC_{\mathcal{A}}$ and $(s(p), a, s'_p) \in \longrightarrow_p$ such that $a = ?c$ implies $\chi(c) \geq 1$. Then configuration $(s', \chi') \in RC_{\mathcal{A}}$ and $((s, \chi), a, (s', \chi')) \in \Longrightarrow_{\mathcal{A}}$, with $s'(p) = s'_p$, $s'(q) = s(q)$ for $q \neq p$, and
  - If $a = !c$, then $\chi'(c) = \chi(c) + 1$ and $\chi'(d) = \chi(d)$ for $d \neq c$.
  - If $a = ?c$, then $\chi'(c) = \chi(c) - 1$ and $\chi'(d) = \chi(d)$ for $d \neq c$.
  - If $a \in \Sigma_p^{cho}$, then $\chi'(c) = \chi(c)$ for $c \in Ch$.

We define s^{fi} to be the global state given by $s^{fi}(p) = s_p^{fi}$ for each p. We term (s^{fi}, χ^0) as the *final* configuration.

We extend $\Longrightarrow_{\mathcal{A}}$ to $RC_{\mathcal{A}} \times \Sigma^\star \times RC_{\mathcal{A}}$ in the obvious way and denote the extension also by $\Longrightarrow_{\mathcal{A}}$. Namely, firstly $(s, \chi) \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} (s, \chi)$ for any (s, χ) in $RC_{\mathcal{A}}$. Secondly, if $(s, \chi) \overset{\sigma}{\Longrightarrow}_{\mathcal{A}} (s', \chi')$ and $(s', \chi') \overset{a}{\Longrightarrow}_{\mathcal{A}} (s'', \chi'')$ where $\sigma \in \Sigma^\star$, $a \in \Sigma$, then $(s, \chi) \overset{\sigma a}{\Longrightarrow}_{\mathcal{A}} (s'', \chi'')$. A *run* of $\mathcal{A}$ is a sequence $\sigma \in \Sigma^*$ such that $(s^{res}, \chi^0) \overset{\sigma}{\Longrightarrow} (s, \chi)$ for some (s, χ) in $RC_{\mathcal{A}}$. We say that σ *ends at* configuration (s, χ), and denote this configuration by (s^σ, χ^σ). We let $Run(\mathcal{A})$ denote the set of runs of $\mathcal{A}$. The run σ is *complete* iff $(s^\sigma, \chi^\sigma) = (s^{fi}, \chi^0)$, and we denote by $Run_{cpl}(\mathcal{A})$ the set of complete runs of $\mathcal{A}$.

Through the rest of this section, we fix a system $\mathcal{A}$. We will often omit $\mathcal{A}$ (e.g. write RC, Run_{cpl} instead of $RC_{\mathcal{A}}, Run_{cpl}(\mathcal{A})$). A configuration (s, χ) in RC is *valid* iff there exists σ with $(s, \chi) \overset{\sigma}{\Longrightarrow} (s^{fi}, \chi^0)$. A run σ is *valid* iff σ ends at a valid configuration. We say that $\mathcal{A}$ is *deadend-free* iff every member of RC is valid. Note that one can effectively decide whether a given system is deadend-free by an easy reduction to the home marking reachability problem of Petri nets [5].

We show in Fig. 1 a system consisting of two processes $P1$ and $P2$ with c and e being buffers directed from $P1$ to $P2$ while o is a buffer directed from $P2$ to

$P1$. The initial states are A and 1 while E and 3 are final states. The sequence $b\,!e\,?e\,!o\,?o$ is a complete run. The run $\sigma = a\,!c\,b\,!e\,?e\,!o\,?o$ is not complete, even though $s^\sigma = (E, 3)$. For, we have $\chi^\sigma(c) = 1 \neq 0$. This system is not deadend-free, since the run σ cannot be extended to a complete run.

2.1 Schedules

Let $(s, \chi) \in RC_{\mathcal{A}}$ be a reachable configuration. We say $a \in \Sigma$ is enabled at (s, χ) iff $(s, \chi) \stackrel{a}{\Longrightarrow} (s', \chi')$ for some (s', χ') in RC. We say that $p \in \mathcal{P}$ is enabled at (s, χ) iff some $a \in \Sigma_p$ is enabled at (s, χ). A *schedule* for $\mathcal{A}$ is a partial function Sch from Run to $\mathcal{P}$ which satisfies the following condition: $Sch(\sigma)$ is defined iff there is some action enabled at (s^σ, χ^σ), and if $Sch(\sigma) = p$, then p is enabled at (s^σ, χ^σ). Notice that if σ is complete, then no action is enabled at (s^σ, χ^σ) and $Sch(\sigma) = \epsilon$. For the schedule Sch, we denote by Run/Sch the set of runs *according to* Sch and define it inductively as follows: $\varepsilon \in Run/Sch$. If $\sigma \in Run/Sch$, $Sch(\sigma) = p$, $a \in \Sigma_p$ and σa is a run, then $\sigma a \in Run/Sch$. In particular, if $Sch(\sigma) = p$ and σ can be extended by two actions a, b of process p, then the schedule must allow both a and b. It is easy to check that this definition of a schedule corresponds to the one in [3].

We say that the schedule Sch is *valid* for $\mathcal{A}$ iff every run in Run/Sch can be extended in $Run/Sch \cap Run_{cpl}$. Next we define $RC/Sch = \{(s^\sigma, \chi^\sigma) \mid \sigma \in Run/Sch\}$, the set of configurations reached via runs according to Sch. We say that Sch is *regular* if RC/Sch is a finite set and Run/Sch is a regular language (in particular, the system under schedule can be described with finite memory). Finally, we say that $\mathcal{A}$ is *quasi-static schedulable* (schedulable for short) iff there exists a *valid and regular* schedule for $\mathcal{A}$. The quasi-static scheduling problem is to determine, given a system $\mathcal{A}$, whether $\mathcal{A}$ is schedulable. Again, it is easy to check that this definition of quasi-static schedulability corresponds to the one in [3]. In particular, the validity of the schedule corresponds to the fact that the system can always answer a query of the environment (by reaching the final configuration).

In the system of Fig. 1, the function $Sch_1(\sigma) = P$ with $P = P1$ if $P1$ is enabled at state (s^σ, χ^σ), $P = P2$ otherwise, is a schedule. However, it is not regular, since $(a\,!c)^* \in Run/Sch_1$ goes through an unbounded number of configurations $(1, A, n, 0, 0)$. On the other hand, the function $Sch_2(\sigma) = P$ with $P = P2$ if $P2$ is enabled at state (s^σ, χ^σ), $P = P1$ otherwise is a valid and regular schedule.

Fig. 1. A task system with two processes $P1, P2$

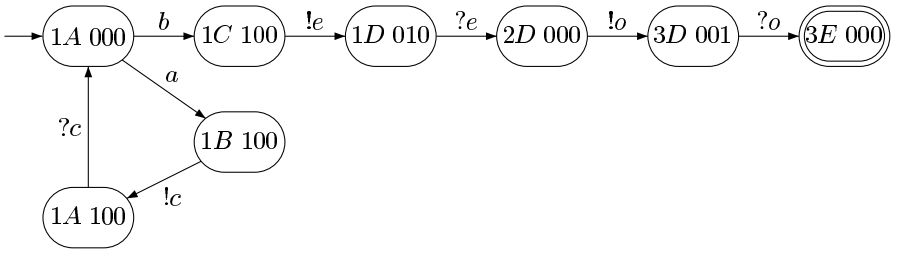


Fig. 2. RC/Sch_2

Fig. 2 shows the finite state space RC/Sch_2 which has no deadend. In this figure, a configuration is of the form $XY\,\alpha\beta\gamma$, with X (Y) the state of $P2$ ($P1$), and α, β, γ denote the contents of buffer c, e and o respectively. That is, the system of Fig. 1 is schedulable. Notice that a schedule does not need to prevent infinite runs. It just must *allow* every run to be completed.

3 General Case and Undecidability

In this section and the next we will only present the main constructions and the proof ideas. Details can be found in [4]. The main result of this section is:

Theorem 1. *The quasi-static scheduling problem is undecidable. In fact, it remains undecidable even when restricted to systems that are deadend-free.*

Given a deterministic two counter machine $\mathcal{M}$, we shall construct a system $\mathcal{A}$ such that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable. More precisely, the constructed $\mathcal{A}$ will have the following property: if *Sch* is a valid schedule for $\mathcal{A}$, then under *Sch* the execution of $\mathcal{A}$ will simulate the execution of $\mathcal{M}$. Further, if the execution of *Sch* leads $\mathcal{A}$ to its final configuration, then in the corresponding execution $\mathcal{M}$ will reach its halting state. We will show that whenever $\mathcal{M}$ halts, $\mathcal{A}$ has a valid schedule *Sch*. Further, *Sch* must lead $\mathcal{A}$ to its final configuration in a finite number of steps, hence it is a valid and regular schedule and $\mathcal{A}$ turns out to be schedulable. On the other hand, if $\mathcal{M}$ does not halt, it will turn out that $\mathcal{A}$ does not even have a valid schedule.

Let C_1, C_2 denote the two counters of $\mathcal{M}$. Let *halt* denote the halting state of $\mathcal{M}$. We assume that, for each control state i other than *halt*, the behaviour of $\mathcal{M}$ at i is given by an instruction in one of the following forms with $j \in \{1, 2\}$:

- $(i, Inc(j), k)$: increment C_j and move to control state k.
- $(i, Dec(j), k, m)$: if $C_j > 0$, then decrement C_j and move to control state k; otherwise ($C_j = 0$), move to control state m.

Thus, $\mathcal{M}$ either stops at *halt* after a finite number of steps, or runs forever without visiting *halt*.

Naturally, we encode counters of $\mathcal{M}$ by buffers of $\mathcal{A}$. Incrementing a counter of $\mathcal{M}$ amounts to sending a data item to the corresponding buffer. And decrementing a counter of $\mathcal{M}$ amounts to picking up a data item from the corresponding buffer. It is clear how the instruction $(i, Inc(j), k)$ of $\mathcal{M}$ can be simulated. The main difficulty is to simulate the instruction $(i, Dec(j), k, m)$. Indeed, in a system, a process can *not* branch to different states according to whether a buffer is empty or not. Further, when a schedule *Sch* selects a process p to execute, *Sch* has to allow all transitions of p that are *enabled* at the current state s_p of p. However, the following observation will facilitate the simulation of an $(i, Dec(j), k, m)$ instruction. Suppose s_p is a polling state with two outgoing transitions labelled $?a$, $?b$, where $src(a) \neq src(b)$. If prior to selecting p and assuming both buffers a and b are currently empty, *Sch* can make the buffer a nonempty (for example, by selecting $src(a)$ to send a data item to a) and keep b empty (for example, by not selecting $src(b)$), then when *Sch* selects p, only the $?a$ transition is enabled and executed, while the $?b$ transition is ignored.

Proof Sketch of Theorem 1: Let $\mathcal{M}$ be a deterministic two-counter machine as above with the associated notations. We construct a system $\mathcal{A}$ such that any *valid* schedule for $\mathcal{A}$ will simulate the execution of $\mathcal{M}$. As discussed above, one can then argue that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable. To ease the presentation, we shall allow a final state to be not a polling state and permit the outgoing transitions of a local state to consist of both receive transitions and choice transitions. It is tedious but easy to modify the transitions of $\mathcal{A}$ so that they strictly adhere to the definition in section 2.

The system $\mathcal{A}$ has five processes $A, C(1), C(2), GD, GZ$. Their communication architecture is illustrated in Fig. 3 where a label ch on an arrow from process p to process q represents a buffer ch with $src(ch) = p$ and $dst(ch) = q$. For $j = 1, 2$, the number of items stored in buffer $c(j)$ will encode the value of counter C_j of $\mathcal{M}$. Process A will mimic the instructions of $\mathcal{M}$. For instructions of the form $(i, Inc(j), k)$, A will need to invoke $C(j)$ to help increment $c(j)$. For instructions of the form $(i, Dec(j), k, m)$, A will invoke GD ("Guess Dec"), GZ ("Guess Zero") so that any valid schedule correctly simulates the emptiness test of buffer $c(j)$. Figure 4 displays the transition systems of GD, GZ, and $C(j)$, $j = 1, 2$, where an initial state is indicated by a pointing arrow, and a final state is drawn as a double circle. Figure 5 illustrates the transition system

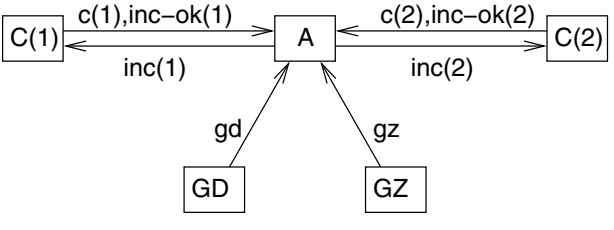


Fig. 3. The architecture of $\mathcal{A}$

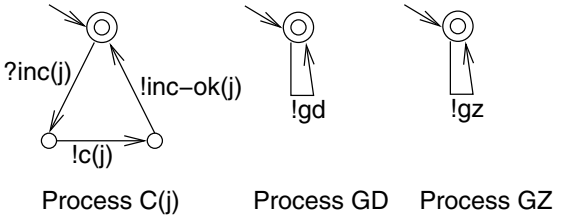


Fig. 4. Description of processes $GD, GZ, C(j)$

of A. For each $(i, Inc(j), k)$ instruction of $\mathcal{M}$, A contains transitions shown in Fig. 5(i). For each $(i, Dec(j), k, m)$ instruction of $\mathcal{M}$, A contains transitions shown in Fig. 5(ii), where the state *sink* is a distinguished state with no outgoing transitions. Unlabelled transitions represent those with labels in Σ^{cho}. For the halting state of $\mathcal{M}$, A contains special transitions shown in Fig. 5(iii), whose purpose is to empty the buffers $c(1), c(2)$ after A reaches *halt*. The initial state of A is the initial state of $\mathcal{M}$, and the final state of A is *halt*.

Let *Sch* be a valid schedule for $\mathcal{A}$. Suppose that, according to *Sch*, execution of $\mathcal{A}$ arrives at a configuration in which A is at state i. There are two cases to consider:

—**Case (i):** The corresponding instruction of $\mathcal{M}$ is $(i, Inc(j), k)$.
It is easy to see that *Sch* will select A to execute $!inc(j)$, then select $C(j)$ three times to execute $?inc(j), !c(j), !inc\text{-}ok(j)$, and finally select A to execute $?inc\text{-}ok(j)$. In doing so, $c(j)$ is incremented and A moves to state k.

—**Case (ii):** The corresponding instruction of $\mathcal{M}$ is $(i, Dec(j), k, m)$.
Note that from state i of A, there are two outgoing transitions labelled $?gd$, $?gz$ respectively. Consider first the case where $c(j)$ is greater than zero. We argue that *Sch* has to guide A to execute *only* the transition $?gd$. That is, *Sch* should ensure that the $?gd$ transition of A is enabled by selecting GD. It must further ensure that the $?gz$ transition of A is *not enabled* which it can do by *not scheduling* the process GZ. By doing so, $c(j)$ will be decremented and A will move to state k. If on the other hand, $?gz$ is enabled while $c(j)$ is greater than zero, then *Sch* will

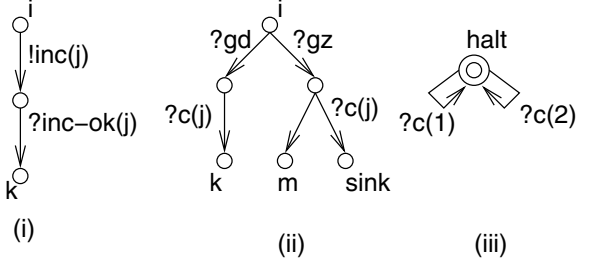


Fig. 5. Transitions of process A

allow A to take the $?gz$ transition. Consequently, *Sch* will allow A to reach state m, *as well as* state *sink*. As *sink* has no outgoing transitions, the run which leads A to *sink* is not valid. This however will contradict the hypothesis that *Sch* is valid.

Similarly, for the case where $c(j)$ is zero, it is easy to see that *Sch* has to guide A to execute *only* $?gz$. Further, after executing the $?gz$ transition, A will move to state m only, since the corresponding $?c(j)$ transition will not be enabled.

We claim that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable. To see this, suppose $\mathcal{M}$ halts. Then from the above argument that $\mathcal{M}$ may be simulated by executing $\mathcal{A}$ under a valid schedule, it is easy to construct a valid schedule *Sch* for $\mathcal{A}$ so that *Sch* will lead $\mathcal{A}$ to the configuration in which each process is at its final state, and all buffers except possibly $c(1), c(2)$ are empty. From Fig. 5(iii), it follows that *Sch* will eventually also empty $c(1), c(2)$. Further, it also follows that *Sch* is regular and thus $\mathcal{A}$ is schedulable.

Suppose $\mathcal{M}$ does not halt. Assume further that *Sch* is a valid schedule for $\mathcal{A}$. Then as explained above, *Sch* simulates the execution of $\mathcal{M}$ and thus process A can never reach its final state *halt*. Thus *Sch* can not be valid, a contradiction. □

4 Data-Branching and Decidability

We have observed that a schedule's ability to indirectly discriminate between two receive actions (e.g. $?gd$ and $?gz$) of the same process is crucial to our undecidability proof. The question arises whether the quasi-static scheduling problem for systems in which such choices are not available is decidable. We show here that the answer is indeed yes. In this context, we wish to emphasize that the definition of quasi static scheduling used in [10] will permit only a finite collection of runs and hence does not cater for systems with internal loops. Thus, the problem solved in [10] is simpler than the one addressed here.

The system $\mathcal{A}$ is said to be *data-branching* iff for each p, for each $s_p \in S_p$, if s_p is a polling state, then it has exactly one outgoing transition. Thus the only branching states are those at which internal data-dependent choices take place.

Theorem 2. *Given a data-branching system $\mathcal{A}$, one can effectively determine whether $\mathcal{A}$ is schedulable.*

The rest of this section is devoted to the proof of theorem 2. We shall assume throughout that $\mathcal{A}$ is data-branching. The proof relies crucially on the notion of a *canonical schedule* for $\mathcal{A}$, denoted Sch_{ca}. The canonical schedule is *positional*, that is, $Sch_{ca}(\sigma) = Sch_{ca}(\sigma')$ whenever runs σ,σ' end at the same configuration. Thus, we shall view Sch_{ca} as a function from RC to $\mathcal{P}$. Informally, at configuration (s, χ), if there is a $p \in \mathcal{P}$ such that p is enabled and s_p is a polling or choice state, then Sch_{ca} picks one such p. If there is no such process, then for each process p enabled at (s, χ), s_p has exactly one outgoing transition $(s_p, !c_p, s'_p)$. In this case, Sch_{ca} picks a process p with $\chi(c_p)$ being minimum. Ties will be broken by fixing a linear ordering on $\mathcal{P}$. The proof of theorem 2 consists of two steps.

Firstly, we show that $\mathcal{A}$ is schedulable iff Sch_{ca} is a valid and regular schedule (Prop. 3). Secondly, we prove that one can effectively decide whether Sch_{ca} is a valid and regular schedule (Thm. 9).

4.1 The Canonical Schedule

We fix a total order $\leq_{\mathcal{P}}$ on $\mathcal{P}$ and define the *canonical schedule* Sch_{ca} for $\mathcal{A}$ as follows. For each configuration (s,χ), let $P^{(s,\chi)}_{enable} \subseteq \mathcal{P}$ be the set of processes enabled at (s,χ). We partition $P^{(s,\chi)}_{enable}$ into $P^{(s,\chi)}_{poll}$, $P^{(s,\chi)}_{choice}$ and $P^{(s,\chi)}_{send}$ as follows. For $p \in P^{(s,\chi)}_{enable}$, we have:(i) $p \in P^{(s,\chi)}_{poll}$ iff s_p is a polling state; (ii) $p \in P^{(s,\chi)}_{choice}$ iff s_p is a choice state; (iii) $p \in P^{(s,\chi)}_{send}$ iff s_p is a sending state. We further define the set $P^{(s,\chi)}_{send\text{-}min} \subseteq P^{(s,\chi)}_{send}$ as follows: for $p \in P^{(s,\chi)}_{send}$, we have $p \in P^{(s,\chi)}_{send\text{-}min}$ iff $\chi(c_p) \leq \chi(c_q)$ for each $q \in P^{(s,\chi)}_{send}$, where $!c_p$ (respectively, $!c_q$) is the action of p (respectively, of q) enabled at (s,χ).

The canonical schedule Sch_{ca} maps each configuration (s,χ) to the process $Sch_{ca}(s,\chi)$ as follows. If $P^{(s,\chi)}_{poll} \cup P^{(s,\chi)}_{choice} \neq \emptyset$, then $Sch_{ca}(s,\chi)$ is the least member of $P^{(s,\chi)}_{poll} \cup P^{(s,\chi)}_{choice}$ with respect to $\leq_{\mathcal{P}}$. Otherwise, $Sch_{ca}(s,\chi)$ is the least member of $P^{(s,\chi)}_{send\text{-}min}$ with respect to $\leq_{\mathcal{P}}$. It is straightforward to verify that Sch_{ca} adheres to the definition of schedule.

Proposition 3. *A data-branching system $\mathcal{A}$ is schedulable iff Sch_{ca} is a valid and regular schedule for $\mathcal{A}$.*

To facilitate the proof of Prop. 3, we introduce now an equivalence on complete runs. For $\sigma \in \Sigma^{\star}$ and $p \in \mathcal{P}$, let $prj_p(\sigma)$ be the sequence obtained from σ by erasing letters not in Σ_p. We define the equivalence relation $\sim \subseteq Run_{cpl} \times Run_{cpl}$ as follows: $\sigma \sim \sigma'$ iff for every $p \in \mathcal{P}$, $prj_p(\sigma) = prj_p(\sigma')$. We note a useful relation between $\sim$ and schedules.

Observation 4. *Let σ be a complete run of a data-branching system $\mathcal{A}$. Suppose that Sch is a schedule of $\mathcal{A}$ (not necessarily valid nor regular). Then there exists a complete run σ' such that $\sigma' \sim \sigma$ and $\sigma' \in Run/Sch$.*

Observation 4 -whose proof can be found in [4]- implies that a run σ of Run/Sch can be extended to a run in Run_{cpl}/Sch iff it can be extended to a run in Run_{cpl}. This holds for every schedule Sch (not necessarily valid nor regular), provided the system is data-branching. Using this observation, we can now prove that if there exists a valid schedule, then Sch_{ca} is valid too.

Lemma 5. *A data-branching system $\mathcal{A}$ admits some valid schedule iff Sch_{ca} is valid for $\mathcal{A}$.*

The concept of an *anchored* path or run, that we introduce now will also play a crucial role in what follows. If χ is a mapping from Ch to the non-negative integers, let $\max(\chi) = \max\{\chi(c) \mid c \in Ch\}$. For a run σ, let $\max(\sigma) = \max\{\max(\chi^{\sigma'}) \mid \sigma'$ is a prefix of $\sigma\}$. We say that σ is an *anchored* run iff $\max(\sigma)$ *is non-null* and

$\max(\sigma) > \max(\chi^{\sigma'})$ for every strict prefix σ' of σ. Anchored runs according to Sch_{ca} have a special property: every action enabled concurrently with and including the last action of an anchored run is a send action on some buffer that holds a maximum number of items. This property may be stated precisely as follows.

Observation 6. *Let σ be an anchored run according to Sch_{ca}, and let $M = \max(\sigma)$. Then $\sigma = \hat{\sigma}!c$ for some $c \in Ch$ and $\chi^{\sigma}(c) = M$. Further, if $a \in \Sigma$ is enabled at $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$, then $a = !d$ for some $d \in Ch$ and moreover $\chi^{\hat{\sigma}}(d) = M - 1$.*

We are now ready to prove Prop. 3.

Proof. of Prop. 3
The if part is obvious. As for the only if part, let Sch be a valid and regular schedule for $\mathcal{A}$. First, it follows from lemma 5 that Sch_{ca} is valid.

We prove that Sch_{ca} is regular. We know that RC/Sch contains a finite number k of configurations. Since each action adds at most one item to one buffer, for all $\sigma \in Run/Sch$, $\max(\sigma) \leq k$. We will prove that for all $\sigma_{ca} \in Run/Sch_{ca}$, $\max(\sigma_{ca}) \leq k$, which will imply that RC/Sch_{ca} has a finite number of configurations. Since we know that Sch_{ca} is valid, it suffices to consider only complete runs of Run/Sch_{ca}.

Let $\sigma_{ca} \in Run/Sch_{ca}$ be a complete run. Following observation 4, let $\sigma \in Run/Sch$ be a complete run such that $\sigma \sim \sigma_{ca}$. Suppose $M_{ca} = \max(\sigma_{ca})$ and $M = \max(\sigma)$. Pick the least prefix τ_{ca} of σ_{ca} such that $\tau_{ca} = M_{ca}$. Thus τ_{ca} is anchored. By observation 6, let $\tau_{ca} = \hat{\tau}_{ca}!c$. Consider the sequence $\hat{\tau}_{ca}$. For a prefix τ of σ, we say τ is *covered* by $\hat{\tau}_{ca}$ iff for every $p \in \mathcal{P}$, $prj_p(\tau)$ is a prefix of $prj_p(\hat{\tau}_{ca})$. Now pick τ to be the least prefix of σ such that τ is *not* covered by $\hat{\tau}_{ca}$. Such a τ exists, following the definition of $\sim$. Let $\tau = \hat{\tau}a$ where $a \in \Sigma$ is the last letter of τ. We consider three cases.

—**Case (i):** $a = !d$ for some $d \in Ch$.
The choice of τ implies $prj_{p_a}(\hat{\tau}) = prj_{p_a}(\hat{\tau}_{ca})$. Thus, $s^{\hat{\tau}}(p_a) = s^{\hat{\tau}_{ca}}(p_a)$. And $!d$ is enabled at configuration $(s^{\hat{\tau}_{ca}}, \chi^{\hat{\tau}_{ca}})$. It follows from observation 6 that $\chi^{\hat{\tau}_{ca}}(d) = M_{ca} - 1$ (whether $d = c$ or not). As $dst(d) \neq p_a$, the choice of τ also implies $prj_{dst(d)}(\hat{\tau})$ is a prefix of $prj_{dst(d)}(\hat{\tau}_{ca})$. Hence, we have $\#_{!d}(\hat{\tau}) = \#_{!d}(\hat{\tau}_{ca})$ and $\#_{?d}(\hat{\tau}) \leq \#_{?d}(\hat{\tau}_{ca})$, where $\#_b(\rho)$ denotes the number of occurrences of letter b in sequence ρ. It follows that $\chi^{\hat{\tau}}(d) \geq \chi^{\hat{\tau}_{ca}}(d)$. Combining these observations with $\chi^{\hat{\tau}}(d) \leq M - 1$ then yields $M_{ca} \leq M$.

–**Case (ii):** $a = ?d$ for some $d \in Ch$.
By the same argument as in case (i), we have $s^{\hat{\tau}}(p_a) = s^{\hat{\tau}_{ca}}(p_a)$. Also we have $prj_{p_a}(\hat{\tau}) = prj_{p_a}(\hat{\tau}_{ca})$, and $prj_{src(d)}(\hat{\tau})$ is a prefix of $prj_{src(d)}(\hat{\tau}_{ca})$. Hence, $\chi^{\hat{\tau}}(d) \leq \chi^{\hat{\tau}_{ca}}$. It follows that $?d$ is enabled at configuration $(s^{\hat{\tau}_{ca}}, \chi^{\hat{\tau}_{ca}})$. This contradicts that at configuration $(s^{\hat{\tau}_{ca}}, \chi^{\hat{\tau}_{ca}})$, the schedule Sch_{ca} picks process $src(c)$ with $s^{\hat{\tau}_{ca}}(src(c))$ being a sending state.

—**Case (iii):** $a \in \Sigma^{cho}_{p_a}$.
Similar to Case (ii), we obtain a contradiction by noting that a is enabled at $(s^{\hat{\tau}_{ca}}, \chi^{\hat{\tau}_{ca}})$. □

4.2 Deciding Boundedness of the Canonical Schedule

The decision procedure for boundedness of Sch_{ca} is similar to the decision procedure for the boundedness of Petri nets [8], but one cannot directly apply [8] because RC/Sch_{ca} cannot be represented as the set of reachable markings of a Petri net. Indeed, the canonical schedule performs a zero-test when it schedules a process ready to send, because it must check that all processes ready to receive have empty input buffers. We show that one can nevertheless build a *finite* tree of configurations in RC/Sch_{ca} that exhibits a witness for unboundedness iff RC/Sch_{ca} is not a finite set or Sch_{ca} is not a valid schedule for $\mathcal{A}$.

Towards this, the following partial order relation on anchored runs will play a useful role. Let Run_{an}/Sch_{ca} be the subset of anchored runs of Run/Sch_{ca}. We define $\prec_{ca} \subseteq Run_{an}/Sch_{ca} \times Run_{an}/Sch_{ca}$ as the least (strict) partial order satisfying the following. For $\sigma, \sigma' \in Run_{an}/Sch_{ca}$, (σ, σ') is in $\prec_{ca}$ whenever $\sigma = \hat{\sigma}!c$, $\sigma' = \hat{\sigma}'!c$ for some $c \in Ch$ and:

- σ is a strict prefix of σ'.
- $s^{\hat{\sigma}}(p) = s^{\hat{\sigma}'}(p)$ for every $p \in \mathcal{P}$.
- $\chi^{\sigma}(d) \leq \chi^{\sigma'}(d)$ for each $d \in Ch$.

Notice that, in particular, $\chi^{\sigma}(c) < \chi^{\sigma'}(c)$ since σ is a strict prefix of σ' and both are anchored. We show now a structural property of $\prec_{ca}$ which will serve us to produce a *finite* coverability tree for all runs. An *infinite* run of $\mathcal{A}$ is an infinite sequence ρ in Σ^{ω} such that every finite prefix of ρ is in $Run(\mathcal{A})$. We say that an infinite run ρ is admitted by Sch_{ca} iff every finite prefix of ρ is admitted by Sch_{ca}.

Proposition 7. *Suppose $\rho \in \Sigma^{\omega}$ is an infinite run admitted by Sch_{ca}. Then there exist two finite prefixes σ,σ' of ρ such that either σ, σ' end at the same configuration, or $\sigma \prec_{ca} \sigma'$ (in which case σ, σ' are both anchored).*

Next we show that any pair of runs σ, σ' with $\sigma \prec_{ca} \sigma'$ witnesses for the unboundedness of RC/Sch_{ca} (or for the non-validity of Sch_{ca}). This requires an argument that differs from [8] because, even though $\sigma' = \sigma\tau$ and both σ, σ' are according to Sch_{ca}, $\sigma\tau^n$ may be incompatible with Sch_{ca} for some n (because of zero-tests). However, we shall argue that if there exist two anchored paths satisfying $\sigma \prec_{ca} \sigma'$ then for every $n = 1, 2, \ldots$, there exists a run ρ_n according to Sch_{ca} such that either $\max(\rho_n) \geq n$ or ρ_n cannot be extended to reach a final configuration.

Proposition 8. *If there exist two anchored paths σ, σ' in Run_{an}/Sch_{ca} such that $\sigma \prec_{ca} \sigma'$, then either RC/Sch_{ca} has an infinite number of configurations or Sch_{ca} is not valid.*

Proof. Suppose $\sigma' = \sigma\tau$. Fix an arbitrary integer $k > 1$ and consider the sequence $\alpha = \sigma\tau\tau \ldots \tau$ (k copies of τ). Following the definition of $\prec_{ca}$, one verifies that α is a run of $\mathcal{A}$. If α cannot be extended to a complete run, then Sch_{ca} is not valid and this ends the proof. Else, by observation 4, there exists a *complete*

run $\rho \sim \alpha w$ which is according to Sch_{ca}, for some $w \in \Sigma^\star$. Let $M = \max(\sigma)$ and $M' = \max(\sigma')$. Let $\sigma = \hat{\sigma}!c$, $\sigma' = \hat{\sigma}'!c$, where $c \in Ch$, $\chi^{\sigma}(c) = M$, $\chi^{\sigma'}(c) = M'$. We show below that $\max(\rho) \geq M + k \cdot (M' - M)$ and thus Sch_{ca} is not regular.

Though $\sigma\tau$ is according to Sch_{ca}, we note that α is not necessarily a prefix of ρ. Let $\alpha = \hat{\alpha}!c$. Consider the sequence $\hat{\alpha}$. For a prefix β of ρ, we say that β is covered by $\hat{\alpha}$ iff for every $p \in \mathcal{P}$, $prj_p(\beta)$ is a prefix of $prj_p(\hat{\alpha})$. Pick β to be the least prefix of ρ such that β is *not* covered by $\hat{\alpha}$. Let $\beta = \hat{\beta}b$ where b is the last letter of β. Let $p_b \in \mathcal{P}$ be the process such that $b \in \Sigma_{p_b}$. The choice of β implies that $prj_{p_b}(\hat{\beta}) = prj_{p_b}(\hat{\alpha})$, and thus $s^{\hat{\beta}}(p_b) = s^{\hat{\alpha}}(p_b)$. Again we consider three cases.

—**Case (i).** $b = !d$ for some $d \in Ch$.
Thus, $!d$ is enabled at configuration $(s^{\hat{\alpha}}, \chi^{\hat{\alpha}})$. Also, as $dst(d) \neq p_b$, we have that $prj_{dst(d)}(\hat{\beta})$ is a prefix of $prj_{dst(d)}(\hat{\alpha})$. Thus, we have $\#_{!d}(\hat{\beta}) = \#_{!d}(\hat{\alpha})$, and $\#_{?d}(\hat{\beta}) \leq \#_{?d}(\hat{\alpha})$, where $\#_a(\theta)$ denotes the number of occurrences of letter a in sequence θ. It follows that $\chi^{\hat{\beta}}(d) \geq \chi^{\hat{\alpha}(d)}$.

Note that $\chi^{\hat{\alpha}}(c) = M + k \cdot (M' - M) - 1$ and $\chi^{\hat{\beta}}(d) \leq \max(\rho) - 1$. Thus, if $d = c$, then we have $\max(\rho) \geq M + k \cdot (M' - M)$. Otherwise, $d \neq c$. By definition of $\prec_{ca}$, we conclude that $!d$ is also enabled at both configurations $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$, $(s^{\hat{\sigma}'}, \chi^{\hat{\sigma}'})$. Thus, we have $\chi^{\hat{\sigma}}(d) = M - 1$, $\chi^{\hat{\sigma}'}(d) = M' - 1$, due to observation 6. It follows that $\chi^{\hat{\alpha}}(d) = M - 1 + k \cdot (M' - M)$. Consequently, we also have $\max(\rho) = M + k \cdot (M' - M)$.

—**Case (ii).** $b = ?d$ for some $d \in Ch$.
Following the definition of $\prec_{ca}$, we have $s^{\hat{\sigma}}(p_b) = s^{\hat{\sigma}'}(p_b) = s^{\hat{\alpha}}(p_b) = s^{\hat{\beta}}(p_b)$. At configuration $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$, Sch_{ca} picks process $src(c)$ where $s^{\hat{\sigma}}(src(c))$ is a sending state. Hence, p_b is not enabled at $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$. That is, $\chi^{\hat{\sigma}}(d) = 0$. Similarly, we have $\chi^{\hat{\sigma}'}(d) = 0$. As a result, $\chi^{\hat{\alpha}}(d) = 0$.

However, by similar arguments as in case (i), one sees that $\#_{?d}(\hat{\beta}) = \#_{?d}(\hat{\alpha})$ and $\#_{!d}(\hat{\beta}) \leq \#_{!d}(\hat{\alpha})$. Thus, $\chi^{\hat{\beta}}(d) \leq \chi^{\hat{\alpha}}(d)$. We obtain a contradiction as $?d$ is enabled at configuration $(s^{\hat{\beta}}, \chi^{\hat{\beta}})$.

—**Case (iii).** $b \in \Sigma^{cho}_{p_b}$.
Similar to Case (ii), we obtain a contradiction by noting that p_b is enabled at $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$. □

The set of all runs of a data-branching system under the canonical schedule Sch_{ca} forms a possibly infinite tree (any data dependent choice performed by a scheduled process induces several branches). Following Karp and Miller's ideas, one may stop exploring this tree whenever coming again to a configuration already visited, or obtaining an anchored run σ' that extends a smaller anchored run σ, i.e. $\sigma \prec_{ca} \sigma'$. Based on this construction of a finite coverability tree, we obtain the following theorem.

Theorem 9. *One can effectively determine whether Sch_{ca} is valid and regular.*

Proof. We construct inductively W, a tree of valid runs admitted by Sch_{ca}. First, ε is in W. Then, suppose that σ is in W and σa is a run admitted by Sch_{ca},

where $a \in \Sigma$. If there exists $\sigma' \in W$ such that $\sigma' \prec_{ca} \sigma a$, then by proposition 8, we can stop the construction of W and report that either Sch_{ca} is not regular or Sch_{ca} is not valid. Otherwise, we check if there exists $\tau \in W$ such that τ ends at the same configuration as σa. If such a τ does not exist, then we add σa to W (otherwise we just ignore σa).

We first prove that the construction of W stops after a finite number of steps. Suppose otherwise. Then members of W form an infinite tree. By König's lemma, there exists an infinite sequence ρ of Σ^ω such that every finite prefix of ρ is in W. Applying proposition 7, we get that there exist two finite prefixes σ,σ' of ρ such that σ is a prefix of σ' and either σ, σ' end at the same configuration or $\sigma \prec_{ca} \sigma'$. In both cases, the construction would not extend σ', hence ρ is not an infinite path, a contradiction.

If the above construction of W terminates without finding $\sigma \prec_{ca} \sigma'$ (reporting that Sch_{ca} is not regular or that Sch_{ca} is not valid), then $\{(s^\sigma, \chi^\sigma) \mid \sigma \in W\}$ is exactly the set of configurations of $Sch_{ca}(RC)$, that is we have the proof that RC/Sch_{ca} is a finite set, and we can test easily whether Sch_{ca} is valid. □

Thm. 2 is now settled by applying Prop. 3 and Thm. 9.

5 Discussion

In this paper, we have considered quasi-static scheduling as introduced in [3] and have provided a negative answer to an open question posed in [9]. Specifically we have shown that for the chosen class of infinite state systems, checking whether a system is quasi-static schedulable is undecidable. We have then identified the data-branching restriction, and proved that the quasi-static scheduling problem is decidable for data-branching systems. Further, our proof constructs both the schedule and the finite state behaviour of the system under schedule. An important concept used in the proof is the canonical schedule that draws much inspiration from the study of existential bounds on channels of communicating systems [6]. In the language of [6], our result can be rephrased as: it is decidable whether a *weak FIFO* data branching communicating system is existentially bounded, when all its local final states are polling states. We recall that the same problem is undecidable [6] for *strong FIFO* communicating systems, even if they are deterministic and deadend free. Our abstraction policy is similar to the one used in [11]. However, we use existential boundedness while [11] checks whether a communicating system is universally bounded, which is an easier notion to check. Note that the canonical schedule may be easily realized in any practical context: it suffices to prevent any process from sending to a buffer that already contains the maximum number of items determined from that schedule. It is worth recalling that these bounds are optimal.

Deadends play an important role in the notion of quasi-static schedulability studied here and previously. However, quasi-static scheduling may stumble on spurious deadends due to the modelling of the task code by an abstract system. The algorithm we have sketched for constructing the canonical schedule may be combined with an iterative removal of spurious deadends. A more ambitious

extension would be to accommodate non data-branching systems. For this purpose, it would be interesting to formulate a notion of quasi-static schedulabilty based purely on existential boundedness and to study decidability issues in this setting.

References

[1] Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. of the ACM 30(2), 323–342 (1983)
[2] Buck, J.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD Dissertation, Berkeley (1993)
[3] Cortadella, J., Kondratyev, A., Lavagno, L., Passerone, C., Watanabe, Y.: Quasi-static scheduling of independent tasks for reactive systems. IEEE Trans. on Comp.-Aided Design 24(10), 1492–1514 (2005)
[4] Darondeau, P., Genest, B., Thiagarajan, P.S., Yang, S.: Quasi-static scheduling of communicating tasks. Technical report, http://www.crans.org/~genest/DGTY08.pdf
[5] de Frutos-Escrig, D.: Decidability of home states in place transition systems. Internal Report. Dpto. Informatica y Automatica. Univ. Complutense de Madrid (1986)
[6] Genest, B., Kuske, D., Muscholl, A.: On communicating automata with bounded channels. Fundamenta Informaticae 80(2), 147–167 (2007)
[7] Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. Int. Federation Information Processing (IFIP) Congress, pp. 471–475 (1974)
[8] Karp, R., Miller, R.: Parallel program schemata. J. Comput. Syst. Sci. 3(2), 147–195 (1969)
[9] Kondratyev, A., Lavagno, L., Passerone, C., Watanabe, Y.: Quasi-static scheduling of concurrent specifications. In: The Embedded Systems Handbook. CRC Press, Boca Raton (2005)
[10] Sgroi, M., Lavagno, L., Watanabe, Y., Sangiovanni-Vincentelli, A.: Quasi-static scheduling of embedded software using equal conflict nets. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 208–227. Springer, Heidelberg (1999)
[11] Leue, S., Mayr, R., Wei, W.: A scalable incomplete test for the boundedness of UML RT models. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 327–341. Springer, Heidelberg (2004)
[12] Liu, C., Kondratyev, A., Watanabe, Y., Sangiovanni-Vincentelli, A.L., Desel, J.: Schedulability Analysis of Petri Nets Based on Structural Properties. In: ACSD 2006, pp. 69–78 (2006)

Strategy Construction for Parity Games with Imperfect Information*

Dietmar Berwanger[1], Krishnendu Chatterjee[2], Laurent Doyen[3], Thomas A. Henzinger[3,4], and Sangram Raje[5]

[1] RWTH Aachen, Germany
[2] CE, University of California, Santa Cruz, U.S.A.
[3] I&C, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[4] EECS, University of California, Berkeley, U.S.A.
[5] IIT Bombay, India

Abstract. We consider *imperfect-information* parity games in which strategies rely on observations that provide imperfect information about the history of a play. To solve such games, i.e., to determine the winning regions of players and corresponding winning strategies, one can use the subset construction to build an equivalent *perfect-information* game. Recently, an algorithm that avoids the inefficient subset construction has been proposed. The algorithm performs a fixed-point computation in a lattice of antichains, thus maintaining a succinct representation of state sets. However, this representation does not allow to recover winning strategies.

In this paper, we build on the antichain approach to develop an algorithm for constructing the winning strategies in parity games of imperfect information. We have implemented this algorithm as a prototype. To our knowledge, this is the first implementation of a procedure for solving imperfect-information parity games on graphs.

1 Introduction

Parity games capture the algorithmic essence of fundamental problems in state-based system analysis [11]. They arise as natural evaluation games for the μ-calculus, an expressive logic that subsumes most specification formalisms for reactive systems, and they are closely related to alternating ω-automata [7].

In the basic variant, a parity game is played on a finite graph with nodes labeled by natural numbers denoting *priorities*. There are two players, Player 1 and Player 2, who take turns in moving a token along the edges of the graph starting from a designated initial node. In a play, the players thus form an infinite path, and Player 1 wins if the least priority that is visited infinitely often is even; otherwise Player 2 wins. These are games of *perfect information*: during the play each of the players is informed about the current position of the token. One key

* This research was supported in part by the NSF grants CCR-0132780, CNS-0720884, and CCR-0225610, by the Swiss National Science Foundation, by the European COMBEST project, and by the Deutsche Forschungsgemeinschaft (DFG).

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 325–339, 2008.

property of parity games is memoryless determinacy: from every initial node, either Player 1 or Player 2 has a winning strategy that does not depend on the history of the play [5]. As a consequence, a winning strategy can be represented as a subset of the edges of the graph, and the problem of constructing a winning strategy is in NP ∩ coNP.

The perfect-information setting is often not sufficient in practice. The need to model uncertainty about the current state of a system arises in many situations. For instance in controller-synthesis applications, certain parameters of the plant under control may not be observable by the controller. Likewise in multi-component design, individual components of a complex system may have private variables invisible to other components. As a way to handle state-explosion problems, one may accept a loss of information in a concrete model in order to obtain a manageable abstract model of imperfect information.

One fundamental question is how to model *imperfect information*. In the classical theory of extensive games, this is done by partitioning the game tree into information sets signifying that a player cannot distinguish between different decision nodes of the same information set [6]. Technically, this corresponds to restricting the set of strategies available to a player by requiring a uniform choice across all nodes of an information set. However, for the algorithmic analysis of games of infinite duration on graphs, the information sets need to be finitely represented. Such a model is obtained by restricting to strategies that rely on observations corresponding to a partitioning of the game graph.

The model of imperfect information games that we consider here was originally introduced in [10]. Like in the perfect-information case, the game is played by two opposing players on a finite graph. The nodes of the graph, called *locations*, are partitioned into information sets indexed by *observations*. Intuitively, the only visible information available to Player 1 during a play is the observation corresponding to the current location, whereas Player 2 has perfect information about the current location of the game. The starting location is known to both players. Following [2], the parity winning condition is defined in terms of priorities assigned to observations.

The basic algorithmic problems about parity games are (1) to determine the *winning region* of a player, that is, the set of initial locations from which he has a winning strategy, and (2) to construct such a *winning strategy*. One straightforward way to solve parity games of imperfect information is based on the following idea [10,2]: after an initial prefix of a play, Player 1 may not know in which precise location the play currently is but, by keeping track of the history, he can identify a minimal set of locations that is guaranteed to contain the current location. Such a set, to which we refer as a *cell*, reflects the knowledge derived by a player from past play. Via a subset construction that associates moves in the game to transitions between cells, the original imperfect-information game over locations is transformed into an equivalent game of perfect information over cells. This approach, however, incurs an exponential increase in the number of states and is therefore inefficient.

For computing the winning region of a game, an algorithm that avoids the explicit subset construction has been proposed recently in [2]. The algorithm exploits a monotonicity property of imperfect-information games: if a cell is winning for Player 1, that is, if he wins from every location of the cell, then he also wins from every subset of the cell. Intuitively, the subcell represents more precise knowledge than the entire cell. It is therefore sufficient to manipulate sets of cells that are *downward-closed* in the sense that, if a cell belongs to the set, then all its subcells also belong to it. As a succinct representation for downward-closed sets of cells, the algorithm maintains antichains that consist of maximal elements in the powerset lattice of cells. The winning region can now be computed symbolically by evaluating its characterization as a μ-calculus formula over the lattice. One particular effect of this procedure is that the discovery of winning cells propagates backwards, rather than forwards from the initial location, and thus avoids the construction and exploration of cells that are not relevant for solving the game.

On many instances, the antichain algorithm performs significantly better than the subset construction for computing winning regions. However, in contrast to the latter, the antichain algorithm does not construct winning strategies. Indeed, we argue that there is no direct way to extract a winning strategy from the symbolic fixed-point computation. In terms of logic, the algorithm evaluates a μ-calculus formula describing the winning region, which corresponds to evaluating a monadic expression with second-order quantifiers that range over (sets of) nodes in the game graph. On the other hand, strategies are not monadic objects; already memoryless location- or observation-based strategies are composed of binary objects, namely, edges of the graph or pairs of cells. In particular, we show that already in parity games of perfect information knowing the winning region of a game does not make the problem of constructing a winning strategy easier. In imperfect-information games there are additional sources of complexity: the size of a winning strategy may be exponentially larger than the winning region, already for reachability objectives. Nevertheless, the construction of winning strategies is crucial for many applications such as controller synthesis or counterexample-guided abstraction-refinement [8].

In this paper, we present an algorithm for constructing winning strategies in parity games of imperfect information. One main concern is to avoid the subset construction. To accomplish this, our algorithm works with symbolic representations of set of cells and builds on the antichain technique. It is based on an elementary algorithm proposed by McNaughton [9] and presented for parity games by Zielonka [13]. This algorithm works recursively: from the viewpoint of Player 1, in each stage a smaller game is obtained by removing the attractor region from which Player 2 can ensure to reach the minimal odd priority. This operation of removal marks the main difficulty in adapting the algorithm to antichains, as the residual subgame is in general not downward-closed. Intuitively, switching between the sides of the two players breaks the succinct representation. We overcome this difficulty by letting, in a certain sense, Player 1 simulate Player 2. Technically, this amounts to replacing two alternating reachability

computations by the computation of a strategy that simultaneously satisfies a reachability and a safety objective.

We have implemented the algorithm as a prototype. To our knowledge, this is the first automatic tool for solving imperfect-information parity games on graphs. A full version of this paper with detailed proofs is available in [1].

2 Definitions

Let Σ be a finite alphabet of actions and let Γ be a finite alphabet of observations. A *game structure of imperfect information* over Σ and Γ is a tuple $G = (L, l_0, \Delta, \gamma)$, where L is a finite set of locations (or states), $l_0 \in L$ is the initial location, $\Delta \subseteq L \times \Sigma \times L$ is a set of labelled transitions, and $\gamma : \Gamma \to 2^L \setminus \emptyset$ is an observability function that maps each observation to a set of locations. Abusing notation, we usually identify the set $\gamma(o)$ with the observation symbol o. We require the following two conditions on G: (i) for all $\ell \in L$ and all $\sigma \in \Sigma$, there exists $\ell' \in L$ such that $(\ell, \sigma, \ell') \in \Delta$, i.e., the transition relation is total, and (ii) the set $\{\gamma(o) \mid o \in \Gamma\}$ partitions L. For each $\ell \in L$, let $\mathsf{obs}(\ell) = o$ be the unique observation such that $\ell \in \gamma(o)$. In the special case where $\Gamma = L$ and $\mathsf{obs}(\ell) = \ell$, for all $\ell \in L$, we say that G is a game structure of *perfect information* over Σ. For infinite sequences of locations $\pi = \ell_1 \ell_2 \ldots$, we define $\mathsf{obs}(\pi) = o_1 o_2 \ldots$ where $\mathsf{obs}(\ell_i) = o_i$ for all $i \geq 1$, and similarly for finite sequences of locations. For $\sigma \in \Sigma$ and $s \subseteq L$, we define $\mathsf{post}_\sigma(s) = \{\ell' \in L \mid \exists \ell \in s : (\ell, \sigma, \ell') \in \Delta\}$ as the set of σ-successors of locations in s.

The game on G is played in rounds. In each round, Player 1 chooses an action $\sigma \in \Sigma$, and Player 2 chooses a successor ℓ' of the current location ℓ such that $(\ell, \sigma, \ell') \in \Delta$. A *play* in G is an infinite sequence $\pi = \ell_1 \ell_2 \ldots$ of locations such that (i) $\ell_1 = l_0$, and (ii) for all $i \geq 0$, there exists $\sigma_i \in \Sigma$ such that $(\ell_i, \sigma_i, \ell_{i+1}) \in \Delta$.

A *strategy* for Player 1 in G is a function $\alpha : \Gamma^+ \to \Sigma$. The set of possible *outcomes* of α in G is the set $\mathsf{Outcome}(G, \alpha)$ of plays $\pi = \ell_1 \ell_2 \ldots$ such that $(\ell_i, \alpha(\mathsf{obs}(\ell_1 \ldots \ell_i)), \ell_{i+1}) \in \Delta$ for all $i \geq 1$. We say that a strategy α is *memoryless* if $\alpha(\rho \cdot o) = \alpha(\rho' \cdot o)$ for all $\rho, \rho' \in \Gamma^*$. We say that a strategy uses *finite memory* if it can be represented by a finite-state deterministic *transducer* $(M, m_0, \lambda, \delta)$ with finite set of states M (the memory of the strategy), initial state $m_0 \in M$, where $\lambda : M \to \Sigma$ labels states with actions, and $\delta : M \times \Gamma \to M$ is a transition function labeled by observations. In state m, the strategy recommends the action $\lambda(m)$, and when Player 2 chooses a location with observation o, it updates the internal state to $\delta(m, o)$. Formally, $(M, m_0, \lambda, \delta)$ defines the strategy α such that $\alpha(\rho) = \lambda(\hat{\delta}(m_0, \rho))$ for all $\rho \in \Gamma^+$, where $\hat{\delta}$ extends δ to sequences of observations in the usual way. The *size* of a finite-state strategy is the number $|M|$ of states of its transducer.

An *objective* for a game structure $G = (L, l_0, \Delta, \gamma)$ is a set $\phi \subseteq \Gamma^\omega$ of infinite sequences of observations. A strategy α for Player 1 is *winning* for an objective ϕ if $\mathsf{obs}(\pi) \in \phi$ for all $\pi \in \mathsf{Outcome}(G, \alpha)$. We say that set of locations $s \subseteq L$ is *winning* for ϕ if there exists a strategy α for Player 1 such that α is winning

for ϕ in $G_\ell := (L, \ell, \Delta, \gamma)$ for all $\ell \in s$. A *game* is a pair (G, ϕ) consisting of a game structure and a matching objective. We say that Player 1 wins the game, if he has a winning strategy for the objective ϕ.

We consider the following classical objectives. Given a set $\mathcal{T} \subseteq \Gamma$ of target observations, the *safety* objective $\mathsf{Safe}(\mathcal{T})$ requires that the play remains within the set $\mathcal{T}$, that is, $\mathsf{Safe}(\mathcal{T}) = \{o_1 o_2 \ldots \mid \forall k \geq 1 : o_k \in \mathcal{T}\}$. Dually, the *reachability* objective $\mathsf{Reach}(\mathcal{T})$ requires that the play visits the set $\mathcal{T}$ at least once, that is, $\mathsf{Reach}(\mathcal{T}) = \{o_1 o_2 \ldots \mid \exists k \geq 1 : o_k \in \mathcal{T}\}$. The *Büchi* objective $\mathsf{Buchi}(\mathcal{T})$ requires that an observation in $\mathcal{T}$ occurs infinitely often, that is, $\mathsf{Buchi}(\mathcal{T}) = \{o_1 o_2 \ldots \mid \forall N \cdot \exists k \geq N : o_k \in \mathcal{T}\}$. Dually, the *coBüchi* objective $\mathsf{coBuchi}(\mathcal{T})$ requires that only observations in $\mathcal{T}$ occur infinitely often. Formally, $\mathsf{coBuchi}(\mathcal{T}) = \{o_1 o_2 \ldots \mid \exists N \cdot \forall k \geq N : o_k \in \mathcal{T}\}$. Finally, given a *priority function* $p : \Gamma \to \mathbb{N}$ that maps each observation to a non-negative integer priority, the *parity* objective $\mathsf{Parity}(p)$ requires that the minimum priority that appears infinitely often is even. Formally, $\mathsf{Parity}(p) = \{o_1 o_2 \ldots \mid \min\{p(o) \mid \forall N \cdot \exists k \geq N : o = o_k\} \text{ is even}\}$. We denote by $\mathsf{coParity}(p)$ the complement objective of $\mathsf{Parity}(p)$, i.e., $\mathsf{coParity}(p) = \{o_1 o_2 \ldots \mid \min\{p(o) \mid \forall N \cdot \exists k \geq N : o = o_k\} \text{ is odd}\}$. Parity objectives are a canonical form to express all ω-regular objectives [12]. In particular, they subsume safety, reachability, Büchi and coBüchi objectives.

Notice that objectives are defined as sets of sequences of observations, and they are therefore visible to Player 1. A game with a safety (resp. reachability) objective defined as a set of plays can be transformed into an equivalent game with a visible safety (resp. reachability) objective in polynomial time.

3 Antichain Algorithm

Let Σ be an alphabet of actions and let Γ be an alphabet of observations. We consider the problem of deciding, given a game structure $G = (L, l_0, \Delta, \gamma)$ and a parity objective ϕ, whether Player 1 has a winning strategy for ϕ in G. If the answer is YES, we ask to construct such a winning strategy. This problem is known to be EXPTIME-complete already for reachability objectives [10,2]. The basic algorithm proposed in [10] constructs a game (G^{K}, ϕ') such that (i) $G^{\mathsf{K}} = (S, s_0, \Delta', \gamma')$ is a game structure of *perfect information* over the action alphabet Σ, and (ii) Player 1 has a winning strategy for ϕ in G if and only if Player 1 has a winning strategy for ϕ' in G^{K}. The game structure G^{K} is obtained by a subset construction where $S = 2^L \setminus \{\emptyset\}$ and $(s_1, \sigma, s_2) \in \Delta'$ if and only if there exists an observation $o \in \Gamma$ such that $s_2 = \mathsf{post}_\sigma(s_1) \cap \gamma(o)$ and $s_2 \neq \emptyset$. In the sequel, we call a set $s \subseteq L$ a *cell*. A cell summarizes the current *knowledge* of Player 1, i.e., the set of possible locations in which the game G can be after the sequence of observations seen by Player 1. Notice that every cell reachable in G^{K} is a subset of some observation, and so the parity objective ϕ' is defined by extending to cells in the natural way the priority function p that defines ϕ. Notice that an objective for G^{K} is a set of infinite sequences of cells, since locations and observations coincide in games of perfect information. In (G^{K}, ϕ'), memoryless winning strategies always exist. Hence, they can be converted into

winning strategies in (G, ϕ) that depend only on the current cell in G^{K}. Due to the explicit construction of G^{K}, this approach involves an exponential blow-up of the original game structure.

In [2], an alternative algorithm is proposed to solve games of imperfect information. Winning cells are computed symbolically, avoiding the exponential subset construction. The algorithm is based on the *controllable predecessor operator* $\mathsf{CPre} : 2^S \to 2^S$ which, given a set of cells q, computes the set of cells q' from which Player 1 can force the game into a cell of q in one round. Formally,

$$\mathsf{CPre}(q) = \{s \in S \mid \exists \sigma \in \Sigma \cdot \forall s' : \text{ if } (s, \sigma, s') \in \Delta' \text{ then } s' \in q\}.$$

The key of the algorithm is that $\mathsf{CPre}(\cdot)$ preserves downward-closedness, which intuitively means that if Player 1 has a strategy from s to force the game to be in q in the next round, then he also has such a strategy from all $s' \subseteq s$ because then Player 1 has a more precise knowledge in s' than in s. Formally, a set q of cells is *downward-closed* if $s \in q$ implies $s' \in q$ for all $s' \subseteq s$. If q is downward-closed, then so is $\mathsf{CPre}(q)$. Since parity games can be solved by evaluating a μ-calculus formula over the powerset lattice $(S, \subseteq, \cup, \cap)$, and since $\mathsf{CPre}(\cdot)$, $\cap$ and $\cup$ preserve downward-closedness, it follows that a symbolic algorithm maintains only downward-closed sets q of cells, and can therefore use a compact representation, namely their maximal elements $\lceil q \rceil = \{s \in q \mid s \neq \emptyset \text{ and } \forall s' \in q : s \not\subset s'\}$, forming *antichains* of cells, i.e., sets of $\subseteq$-incomparable cells. The set $\mathcal{A}$ of antichains is partially ordered as follows: for $q, q' \in \mathcal{A}$, let $q \sqsubseteq q'$ iff $\forall s \in q \cdot \exists s' \in q' : s \subseteq s'$. The least upper bound of $q, q' \in \mathcal{A}$ is $q \sqcup q' = \lceil \{s \mid s \in q \text{ or } s \in q'\} \rceil$, and their greatest lower bound is $q \sqcap q' = \lceil \{s \cap s' \mid s \in q \text{ and } s' \in q'\} \rceil$. The partially ordered set $(\mathcal{A}, \sqsubseteq, \sqcup, \sqcap)$ forms a complete lattice. We view antichains of location sets as a symbolic representation of $\subseteq$-downward-closed sets of cells.

The advantage of the symbolic antichain approach over the explicit subset construction has been established in practice for different applications in model-checking (*e.g.* [3,4]). The next lemma shows that the antichain algorithm may be exponentially faster than the subset construction.

Lemma 1 (See also [3]). *There exists a family $(G_k)_{k \geq 2}$ of reachability games of imperfect information with k locations such that, on input G_k the subset-construction algorithm runs in time exponential in k whereas the antichain algorithm runs in time polynomial in k.*

The antichain algorithm computes a compact representation of the set of winning cells. However, it does not produce a winning strategy. We point out that, already for parity games with perfect information, if there exists a polynomial-time algorithm that, given a game and the set of winning locations for Player 1, constructs a memoryless winning strategy, then parity games can be solved in polynomial time.

Proposition 2. *The following two problems on parity games with perfect information in which Player 1 wins are polynomial-time equivalent.*

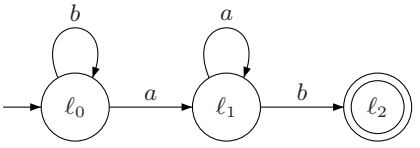


Fig. 1. A reachability game G

(i) Given a game, construct a memoryless winning strategy.
(ii) Given a game and the set of winning locations for Player 1*, construct a memoryless winning strategy.*

Proof. For any instance of problem (i), that is, a game G where Player 1 wins from the initial location l_0, we construct an instance (G', W) of problem (ii) in such a way that every memoryless winning strategy in G' corresponds to a winning strategy for G. (The converse is trivial.)

Without loss, we assume that no priorities in G are less than 2. The game G' is obtained by adding to G a "reset" location z of priority 1, with transitions that allow Player 1 to reach z from any location of G where he moves, and with one transition from z back to l_0. In the new game, Player 1 wins from any location by first moving via z to l_0 and then following the winning strategy he has in G. Thus, G' together with the set of all locations is an instance of problem (ii). Obviously this can be constructed in polynomial time. Let now α be a memoryless winning strategy in G'. No play starting from l_0 that follows α can reach z, otherwise Player 1 loses. Thus, α is readily a winning strategy in the original game G. ■

We also argue that, in games with imperfect information, even for simple reachability objectives the antichain representation of the set of winning cells may not be sufficient to construct a winning strategy. Consider the game G depicted in Fig. 1, with reachability objective $\mathsf{Reach}(\{\ell_2\})$. The observations are $\{\ell_0, \ell_1\}$ and $\{\ell_2\}$. Since $\mathsf{CPre}(\{\{\ell_2\}\}) = \{\{\ell_1\}\}$ (by playing action b) and $\mathsf{CPre}(\{\{\ell_1\}, \{\ell_2\}\}) = \{\{\ell_0, \ell_1\}\}$ (by playing action a), the fixed-point computed by the antichain algorithm is $\{\{\ell_2\}, \{\ell_0, \ell_1\}\}$. However, from $\{\ell_0, \ell_1\}$, after playing a, Player 1 reaches the cell $\{\ell_1\}$ which is not in the fixed-point (however, it is subsumed by the cell $\{\ell_0, \ell_1\}$). Intuitively, the antichain algorithm has forgotten which action is to be played next. Notice that playing a again, and thus forever, is not winning. The next lemma formalizes this intuition.

Lemma 3. *There exists a family of games G_k with $O(p(k))$ many locations for a polynomial p, and a reachability objective ϕ, such that the fixed point computed by the antichain algorithm for (G_k, ϕ) is of polynomial size in k, whereas any finite-memory winning strategy for (G_k, ϕ) is of exponential size in k.*

We first present the ingredients of the proof informally. Let $p_1, p_2, \ldots$ be the list of prime numbers in increasing order. For $k \geq 1$, let $\Sigma_k = \{1, \ldots, k\}$. The action alphabet of the game G_k is $\Sigma_k \cup \{\#, \bot\}$. The game is composed of subgames H_i, each consisting of a loop over p_i many locations $\ell_1, \ldots, \ell_{p_i}$. From a location

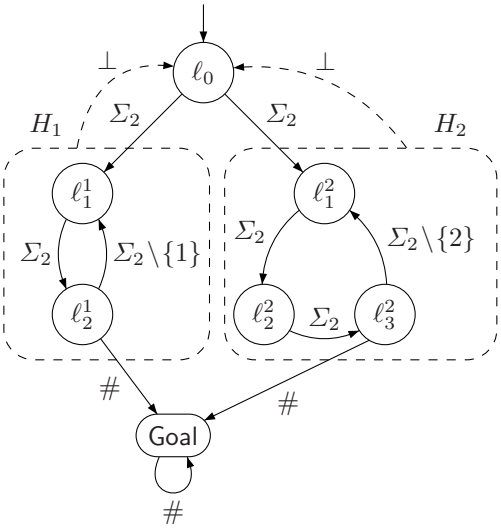


Fig. 2. The game G_2

ℓ_j all actions in Σ_k lead to ℓ_{j+1} and from the last location ℓ_{p_i} Player 1 can return to the initial location ℓ_1 with any action in Σ_k except i. Formally, for all $1 \leq i \leq k$, we define the subgame H_i with location space $L_i = \{\ell_1, \ldots, \ell_{p_i}\}$, initial location ℓ_1, and transition relation $E_i = \{(\ell_j, \sigma, \ell_{j+1}) \mid 1 \leq j \leq p_i - 1 \wedge \sigma \in \Sigma_k\} \cup \{(\ell_{p_i}, \sigma, \ell_1) \mid \sigma \in \Sigma_k \setminus \{i\}\}$. In the sequel, we assume that the location spaces of all H_i are disjoint, *e.g.* by adding a superscript i to the locations of L_i ($L_i = \{\ell_1^i, \ldots, \ell_{p_i}^i\}$).

Fig. 2 shows the game G_k for $k = 2$. In general, in G_k, there is a unique trivial observation, so it is a blind game. We also assume that playing a particular action in a location where it is not allowed leads to a sink location from which Goal is not reachable. The plays start in location ℓ_0 where every move in Σ_k is allowed. The next location can be any of the initial locations of the subgames H_i. Thus, Player 1 can henceforth play any action $\sigma \in \Sigma_k$, except in the last location ℓ_{p_i} where playing $\sigma = i$ would lead to the sink. As he does not know in which of the H_i the play currently is, he should avoid playing $\sigma = i$ whenever his knowledge set contains $q_{p_i}^i$. However, after a certain number of steps (namely $p_k^* = \prod_{i=1}^k p_i$), the current location of the game will be one of the $\ell_{p_i}^i$. Then, taking a transition labeled by # necessarily leads to Goal. The # is not allowed in any other location, so that Player 1 needs to count the first p_k^* steps before playing that move. Notice that after the first round, Player 1 could play $\bot$, but this would not reduce the amount of memory needed to win. However, it shows that he is winning uniformly from all locations of the subgames H_i. Since the size p_k^* of the strategy is exponential in the size $\sum_{i=1}^k p_i$ of the game, the theorem follows.

Proof of Lemma 3. The location space of G_k is the disjoint union of $L_1, \ldots, L_k$ and $\{q_0, \mathsf{Goal}, \mathsf{Bad}\}$. The initial location is q_0, the target observation consists of

Goal, and the sink location is Bad. The transition relation contains each set E_i, the transitions $(\ell^i_j, \bot, \ell_0)$, and the transitions $(\ell_0, \sigma, \ell^i_1)$ and $(\ell^i_{p_i}, \#, \mathsf{Bad})$ for all $1 \leq i \leq k$, $1 \leq j \leq p_i$ and $\sigma \in \Sigma_k$. The transition relation is made total by adding the transitions $(q, \sigma, \mathsf{Bad})$ for each location ℓ of G_n and $\sigma \in \Sigma_k \cup \{\#\}$ such that there is no transition of the form (q, σ, q') for $q' \neq \mathsf{Bad}$. There is only one trivial observation, i.e., the observation alphabet Γ is a singleton.

First we show that Player 1 wins G_k. As there is exactly one observation, a strategy for Player 1 is a function $\lambda : \mathbb{N}^{\geq 0} \to \Sigma_k \cup \{\#, \bot\}$. We define the sets S_j such that any strategy λ such that $\lambda(j) \in S_j$ for all $j \geq 1$ is winning for Player 0. We take $S_1 = \Sigma_k$, $S_j = \{i \in \Sigma_k \mid j - 1 \mod p_i \neq 0\}$ for $2 \leq j \leq p^*_k$. Notice that $S_j \neq \emptyset$ because the least common multiple of $p_1, \ldots, p_k$ is p^*_k. Finally, for $j > p^*_k$ we take $S_j = \{\#\}$. It is easy to show that any strategy defined by these sets is winning for Player 1.

For the second part of the theorem assume, towards a contradiction, that there exists a finite-state winning strategy $\hat{\lambda}$ with less than p^*_k states. Clearly, when playing any winning strategy, the $(p^*_k + 1)$-th location of the play in G_k must be $\ell^i_{p_i}$ for some $i \in \{1, \ldots, k\}$. Moreover, each of the states $\ell^i_{p_i}$ could be the current one, depending on the initial choice of Player 2 (after the first move of Player 1). Therefore, after p^*_k steps, any winning strategy must play $\#$. In the case of $\hat{\lambda}$, the state of the automaton for $\hat{\lambda}$ after p^*_k steps has necessarily been visited in one of the previous steps. This means that $\#$ has been played before and thus $\hat{\lambda}$ is not a winning strategy as for all $j < p^*_k$, one of the subgames H_i is not in location $\ell^i_{p_i}$ after j steps of play, and thus playing $\#$ leads to a loss for Player 1. ■

Finally, we show that it is not trivial to efficiently compute $\mathsf{CPre}(\cdot)$. In the antichain representation, the controllable predecessor operator is defined as

$$\mathsf{CPre}(q) = \left\lceil \{s \subseteq L \mid \exists \sigma \in \Sigma \cdot \forall o \in \Gamma \cdot \exists s' \in q : \mathsf{post}_\sigma(s) \cap \gamma(o) \subseteq s'\} \right\rceil,$$

or equivalently as

$$\mathsf{CPre}(q) = \bigsqcup_{\sigma \in \Sigma} \prod_{o \in \Gamma} \bigsqcup_{s' \in q} \{\widetilde{\mathsf{pre}}_\sigma(s' \cup \overline{\gamma(o)})\}, \tag{1}$$

where $\widetilde{\mathsf{pre}}_\sigma(s) = \{s' \in S \mid \mathsf{post}_\sigma(\{s'\}) \subseteq s\}$ and $\overline{\gamma(o)} = L \setminus \gamma(o)$.

Notice that the least upper bound of a set $\{\ell_1, \ldots, \ell_k\}$ of antichains can be computed in polynomial time, whereas a naive algorithm for the greatest lower bound is exponential. The next lemma shows that, as long as we use a reasonable representation of antichains which allows to decide in polynomial time whether an antichain contains a set larger than n, it is unlikely that $\mathsf{CPre}(\cdot)$ is computable in polynomial time.

Lemma 4. *The following problem is* NP-HARD*: given a game of imperfect information G, an antichain q and an integer n, decide whether there exists a set $B \in \mathsf{CPre}(q)$ with $|B| \geq n$.*

4 Strategy Construction with Antichains

We present a procedure to construct a winning strategy for a parity game of imperfect information $G = (L, l_0, \Delta, \gamma)$ over the alphabets Σ and Γ. It is sometimes convenient to reason in terms of the equivalent perfect-information game G^{K} obtained via the subset construction in Section 3. Let $\mathcal{C}$ denote the set of all cells s such that $s \subseteq \gamma(o)$ for some $o \in \Gamma$. Thus, $\mathcal{C}$ contains all locations of G^{K}. For $\mathcal{R} \subseteq \mathcal{C}$, a *cell strategy* on $\mathcal{R}$ is a memoryless strategy $\alpha : \mathcal{R} \to \Sigma$ for Player 1 in G^{K}. Given an objective $\phi \subseteq \mathcal{C}^\omega$ in G^{K}, we define

$$\mathsf{Win}^{\mathcal{R}}(\phi) := \{\, s \in \mathcal{R} \mid \text{ there exists a cell strategy } \alpha \text{ such that } \mathsf{Outcome}(G^{\mathsf{K}}_s, \alpha) \subseteq \phi \cap \mathsf{Safe}(\mathcal{R}) \,\}.$$

In words, $\mathsf{Win}^{\mathcal{R}}(\phi)$ consists of cells s such that given the initial cell is s there exists a winning cell strategy for Player 1 to ensure ϕ while maintaining the game G^{K} in $\mathcal{R}$.

In Algorithm 1, we present a procedure to construct a winning cell strategy in G^{K} for objectives of the form

$$\mathsf{Reach}(\mathcal{T}) \cup (\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F})),$$

where $\mathcal{T}, \mathcal{F} \subseteq \mathcal{C}$ are downward-closed sets of cells and $p : \Gamma \to \mathbb{N}$ is a priority function over observations. As p can be naturally extended to cells, the set $\mathsf{Parity}(p)$ contains the sequence of cells such that the minimal priority cell appearing infinitely often is even. The parity objective $\mathsf{Parity}(p)$ corresponds to the special case where $\mathcal{F} = \mathcal{C}$ and $\mathcal{T} = \emptyset$. Note that a winning strategy need not be defined on $\mathcal{T}$ since $\mathsf{Reach}(\mathcal{T})$ is satisfied for all cells in $\mathcal{T}$. Memoryless strategies are sufficient for this kind of objective in games of perfect information. Thus, we can restrict our attention without loss to memoryless cell strategies.

Informal description. The algorithm is based on two procedures $\mathsf{ReachOrSafe}(\mathcal{T}, \mathcal{F})$ and $\mathsf{ReachAndSafe}(\mathcal{T}, \mathcal{F})$ that use antichains to compute the set of winning cells and a winning strategy for the objectives $\mathsf{Reach}(\mathcal{T}) \cup \mathsf{Safe}(\mathcal{F})$ and $\mathsf{Reach}(\mathcal{T}) \cap \mathsf{Safe}(\mathcal{F})$, respectively, given downward-closed sets of cells $\mathcal{T} \subseteq \mathcal{C}$ and $\mathcal{F} \subseteq \mathcal{C}$. For perfect-information games, it is known that memoryless winning strategies exist for such combinations of safety and reachability objectives.

The procedure is called recursively, reducing the number of priorities. Given a parity function p we denote by $p-2$ the parity function such that for all $o \in \Gamma$ we have $(p-2)(o) = p(o)$ if $p(o) \leq 1$, and $(p-2)(o) = p(o) - 2$ otherwise. For $i \geq 0$, we denote by $\mathcal{C}_p(i) = \{\, s \in \mathcal{C} \mid s \subseteq \gamma(o), o \in \Gamma, p(o) = i \,\}$ the set of cells with priority i. Let W_1 and W_2 be disjoint sets of cells, and let α_1 be a cell strategy on W_1 and α_2 be a cell strategy on W_2. We denote by $\alpha_1 \cup \alpha_2$ the cell strategy on $W_1 \cup W_2$ such that for all $s \in W_1 \cup W_2$, we have $(\alpha_1 \cup \alpha_2)(s) = \alpha_1(s)$ if $s \in W_1$, and $(\alpha_1 \cup \alpha_2)(s) = \alpha_2(s)$ otherwise.

Without loss of generality we assume that the cells in the target set $\mathcal{T}$ are *absorbing* (i.e., have self-loops only). In line 1 of Algorithm 1, we compute $W = \mathsf{Win}^{\mathcal{C}}(\phi)$ using the antichain algorithm of [2]. Since we assume that cells in $\mathcal{T}$ are

Algorithm 1. Imperfect-Information Game Solver - **Solve**(G, $\mathcal{T}$, $\mathcal{F}$, p)

Input : A game structure G with target $\mathcal{T} \subseteq \mathcal{C}$, safe set $\mathcal{F} \subseteq \mathcal{C}$ and parity function p on Γ.

Output : $W = \mathsf{Win}^{\mathcal{C}}(\phi)$ where $\phi := \mathsf{Reach}(\mathcal{T}) \cup (\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F}))$, and a winning cell strategy α on $W \setminus \mathcal{T}$ for ϕ.

begin

1 $W \leftarrow \mathsf{Win}^{\mathcal{C}}(\phi)$
2 $(W^*, \alpha^*) \leftarrow \mathsf{ReachAndSafe}(\mathcal{T}, W)$
3 $(W_0, \alpha_0) \leftarrow \mathsf{ReachAndSafe}(W^* \cup (\mathcal{C}_p(0) \cap W), W))$
4 Let α_0' be a cell strategy on $(\mathcal{C}_p(0) \cap W) \setminus W^*$ such that
5 $\quad \mathsf{post}_{\alpha_0'(s)}(s) \cap \gamma(o) \in W$ for all $o \in \Gamma$ and $s \in (\mathcal{C}_p(0) \cap W) \setminus W^*$
6 $\overline{\alpha}_0 \leftarrow \alpha_0 \cup \alpha_0' \cup \alpha^*$
7 $i \leftarrow 0$
8 **repeat**
9 $\quad A_i \leftarrow W \setminus W_i$
10 $\quad$ **if** $W \subseteq \mathcal{C}_p(0) \cup \mathcal{C}_p(1) \cup \mathcal{C}_p(2)$ **then**
11 $\quad\quad (W_{i+1}, \alpha_{i+1}) \leftarrow \mathsf{ReachOrSafe}(W_i, A_i \cap \mathcal{C}_p(2))$
12 $\quad$ **else**
$\quad\quad (W_{i+1}, \alpha_{i+1}) \leftarrow$ **Solve**(G, W_i, $A_i \setminus \mathcal{C}_p(1)$, $p-2$)
13 $\quad \overline{\alpha}_{i+1} \leftarrow \overline{\alpha}_i \cup \alpha_{i+1}$
14 $\quad i \leftarrow i+1$
$\quad$ **until** $W_i = W_{i-1}$
15 $\quad$ **return** $(W_i, \overline{\alpha}_i)$

end

absorbing, a winning cell strategy for the objective ϕ ensures that the set W is never left. In the rest of the algorithm and in the arguments below, we consider the sub-game induced by W. In line 2, the set W^* of winning cells and a winning cell strategy α^* on $W^* \setminus \mathcal{T}$ for the objective $\mathsf{Reach}(\mathcal{T})$ is computed by invoking the procedure $\mathsf{ReachOrSafe}$ with target $\mathcal{T}$ and safe set W. Then the set W_0 of cells is obtained along with a cell strategy α_0 that ensures that either W^* is reached or the set of priority 0 cells in W is reached. After this, the algorithm iterates a loop as follows: at iteration $i+1$, let W_i be the set of cells already obtained by the previous iteration and let $A_i = W \setminus W_i$. The algorithm is invoked recursively with W_i as target set, $A_i \setminus \mathcal{C}_p(1)$ as the safe set, and $p-2$ as the priority function to obtain a set W_{i+1} as a result. In the base case, where W consists of priorities 0, 1 and 2 only, since A_i has no priority 0 cells, the objective $\mathsf{Reach}(W_i) \cup (\mathsf{Parity}(p-2) \cap \mathsf{Safe}(A_i \setminus \mathcal{C}_p(1))$ can be equivalently written as $\mathsf{Reach}(W_i) \cup \mathsf{Safe}(A_i \cap \mathcal{C}_p(2))$. Therefore, in the base case, the recursive call is replaced by $\mathsf{ReachOrSafe}(W_i, A_i \cap \mathcal{C}_p(2))$. Notice that $W_i \subseteq W_{i+1}$. The algorithm proceeds until a fixpoint of $W_i = W_{i+1}$ is reached.

Correctness of the iteration. First, we have $W \setminus W^* \subseteq \mathcal{F}$ which essentially follows from the fact that from $W \setminus W^*$ Player 1 cannot reach $\mathcal{T}$. More

precisely, if a cell $s \in W \setminus W^*$ does not belong to $\mathcal{F}$, then against every cell strategy for Player 1, there is a Player 2 strategy to ensure that the set $\mathcal{T}$ is not reached from s. Hence from s, against every cell strategy for Player 1, there is a Player 2 strategy to ensure that $\mathsf{Reach}(\mathcal{T}) \cup \mathsf{Safe}(\mathcal{F})$ is violated, and thus $\phi = \mathsf{Reach}(\mathcal{T}) \cup (\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F}))$ is violated. This contradicts $s \in W = \mathsf{Win}^{\mathcal{C}}(\phi)$. The significance of the claim is that if W^* is reached, then Player 1 can ensure that $\mathcal{T}$ is reached, and since $W \setminus W^* \subseteq \mathcal{F}$ it follows that if W^* is not reached then the game stays safe in $\mathcal{F}$.

To establish the correctness of the iterative step, we claim that from the set W_{i+1} the cell strategy α_{i+1} on $W_{i+1} \setminus W_i$ which ensures

$$\mathsf{Reach}(W_i) \cup \big(\mathsf{Parity}(p-2) \cap \mathsf{Safe}(A_i \setminus \mathcal{C}_p(1))\big),$$

also ensures that

$$\mathsf{Reach}(W_i) \cup \big(\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F} \setminus \mathcal{C}_p(1))\big).$$

Notice that in $A_i \setminus \mathcal{C}_p(1)$, there is no cell with priority 0 or priority 1 for the priority function p since $\mathcal{C}_p(0) \cap W \subseteq W_0 \subseteq W_i$. Hence, we have

$$\mathsf{Parity}(p-2) \cap \mathsf{Safe}(A_i \setminus \mathcal{C}_p(1)) = \mathsf{Parity}(p) \cap \mathsf{Safe}(A_i \setminus \mathcal{C}_p(1)).$$

Since $A_i \subseteq W \setminus W_0 \subseteq W \setminus W^* \subseteq \mathcal{F}$, it follows that the cell strategy α_{i+1} on $W_{i+1} \setminus W_i$ to ensure

$$\mathsf{Reach}(W_i) \cup \big(\mathsf{Parity}(p-2) \cap \mathsf{Safe}(A_i \setminus \mathcal{C}_p(1))\big),$$

also ensures that

$$\mathsf{Reach}(W_i) \cup \big(\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F} \setminus \mathcal{C}_p(1))\big).$$

holds from all cells in W_{i+1}. By induction on i, composing the cell strategies (i.e., by taking the union of strategies obtained in the iteration) we obtain that from W_{i+1}, the cell strategy $\overline{\alpha}_{i+1}$ on $W_{i+1} \setminus \mathcal{T}$ for Player 1 ensures $\mathsf{Reach}(W_0) \cup \big(\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F}) \cap \mathsf{coBuchi}(\mathcal{F} \setminus \mathcal{C}_p(1))\big)$. Note that to apply the induction step for i times, one may visit cells in $\mathcal{C}_p(1)$, but only finitely many times.

Termination. We claim that upon termination, we have $W_i = W$. Assume towards a contradiction that the algorithm terminates with $W_i = W_{i+1}$ and $W_{i+1} \neq W$. Then the following assertions hold. The set $A_i = W \setminus W_i$ is nonempty and

$$W_{i+1} = W_i = \mathsf{Win}^W\big(\mathsf{Reach}(W_i) \cup \big(\mathsf{Parity}(p-2) \cap \mathsf{Safe}(A_i \setminus \mathcal{C}_p(1))\big)\big),$$

that is, in the whole set A_i against all Player 1 cell strategies, Player 2 can ensure the complementary objective, i.e.,

$$\mathsf{Safe}(A_i) \cap \big(\mathsf{coParity}(p-2) \cup \mathsf{Reach}(A_i \cap \mathcal{C}_p(1))\big).$$

Now, we show that satisfying the above objective also implies satisfying $\mathsf{Safe}(A_i) \cap \mathsf{coParity}(p)$. Consider a cell strategy for Player 1, and consider the counter-strategy for Player 2 that ensures that the game stays in A_i, and also ensures that $\mathsf{coParity}(p-2) \cup \mathsf{Reach}(A_i \cap \mathcal{C}_p(1))$ is satisfied. If a play visits $A_i \cap \mathcal{C}_p(1)$ only finitely many times, then from some point onwards it only visits cells in A_i that do not have priority 1 or priority 0 for the priority function p, and then $\mathsf{coParity}(p-2) = \mathsf{coParity}(p)$. Otherwise, the set $A_i \cap \mathcal{C}_p(1)$ is visited infinitely often and A_i is never left. Since A_i has no 0 priority cells for the priority function p, it means that Player 2 satisfies the $\mathsf{coParity}(p)$ objective. It follows that in A_i against all Player 1 cell strategies, Player 2 can ensure $\mathsf{Safe}(A_i) \cap \mathsf{coParity}(p)$. This is a contradiction to the fact that $A_i \subseteq W = \mathsf{Win}^W(\phi)$ and $\mathsf{Safe}(A_i) \cap \mathsf{coParity}(p) \subseteq \Gamma^\omega \setminus \phi$. This leads to the following theorem.

Theorem 5. *Given an imperfect-information game G with target $\mathcal{T} \subseteq \mathcal{C}$, safe set $\mathcal{F} \subseteq \mathcal{C}$ and a parity function p on Γ, Algorithm 1 computes $W = \mathsf{Win}^{\mathcal{C}}(\phi)$, where $\phi = \mathsf{Reach}(\mathcal{T}) \cup (\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F}))$, and a winning cell strategy α on $W \setminus \mathcal{T}$ for ϕ.*

Proof. This follows from the correctness of the iteration, and the fact $W = W_i$ for some i, it follows that from all locations in W, the obtained cell strategy ensures

$$\mathsf{Reach}(W_0) \cup (\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F}) \cap \mathsf{coBuchi}(\mathcal{F} \setminus \mathcal{C}_p(1)).$$

We now complete the argument by showing that the cell strategy is winning for ϕ. The cell strategy on W_0 ensures that $\mathcal{T}$ is reached from cells in W^*, from cells in $\mathcal{C}_p(0) \cap W$ it ensures to stay in W, and in all remaining cells in W_0 it ensures to reach $W^* \cup (\mathcal{C}_p(0) \cap W)$. The following case analysis completes the proof.

1. If the set W_0 is visited infinitely often, then (a) if W^* is reached, then $\mathcal{T}$ is reached; (b) otherwise $\mathcal{C}_p(0) \cap W$ is visited infinitely often and the game always stays safe in $W \setminus W^* \subseteq \mathcal{F}$. This ensures that $\mathsf{Parity}(p)$ is also satisfied.

2. If W_0 is visited only finitely often, then the play never reaches W^*, otherwise it would reach $\mathcal{T}$ and stay in $\mathcal{T}$ forever, and hence $\mathsf{Safe}(\mathcal{F})$ is satisfied, such that the objective $\mathsf{Parity}(p) \cap \mathsf{Safe}(\mathcal{F}) \cap \mathsf{coBuchi}(\mathcal{F} \setminus \mathcal{C}_p(1))$ is attained. Overall, it follows the objective ϕ is satisfied. ■

Antichain algorithm. To turn Algorithm 1 into an antichain algorithm, all set operations must preserve the downward-closed property. The union and intersection operations on sets preserve the downward-closed property of sets, but the complementation operation does not. Observe that Algorithm 1 performs complementation in line 9 ($A_i \leftarrow W \setminus W_i$) and uses the set A_i in lines 11 and 12. This was done for the ease of correctness proof of the algorithm. To see that the complementation step is not necessary, observe that

$$\mathsf{Reach}(W_i) \cup \big(\mathsf{Parity}(p-2) \cap \mathsf{Safe}(A_i \setminus \mathcal{C}_p(1))\big) = \mathsf{Reach}(W_i) \cup \big(\mathsf{Parity}(p-2) \cap \mathsf{Safe}(W \setminus \mathcal{C}_p(1))\big).$$

Indeed, if a play never visits W_i, then the play is in $\mathsf{Safe}(A_i \setminus \mathcal{C}_p(1))$ if, and only if, it is in $\mathsf{Safe}(W \setminus \mathcal{C}_p(1))$. Also note that the expression $\mathsf{Parity}(p-2) \cap \mathsf{Safe}(W \setminus \mathcal{C}_p(1))$ can be equivalently written as $\mathsf{Parity}(p-2) \cap \mathsf{Safe}(W \cap \bigcup_{i \geq 2} \mathcal{C}_p(i))$. It follows that every set operation in Algorithm 1 preserves downward-closed property. This demonstrates the following statement.

Theorem 6. *Algorithm 1 is compatible with the antichain representation.*

We remark that the explicit construction of the strategies takes place only in few steps of the algorithm: at line 2 and 3 of each recursive call where cell strategies are computed for reachability objectives, and in the base case (parity games with priorities $0, 1$ and 2) in line 11 where cell strategies are computed for union of safety and reachability objectives. Also note that we never need to compute strategies for the target set $\mathcal{T}$, and therefore in line 10, we would obtain strategies for the set $W_{i+1} \setminus W_i$. Hence, once the strategy is computed for a set, then it is never modified in any subsequent iteration.

5 Implementation

We have implemented Algorithm 1 in a prototype written in C. The input is a text-file description of the game structure, transitions and observations. Internally, transitions and sets of locations are represented as arrays of integers.

The building blocks of the algorithm are the computation of $\mathsf{CPre}(\cdot)$, and the two procedures ReachOrSafe and ReachAndSafe. The implementation for $\mathsf{CPre}(q)$ follows Equation (1) using three nested loops over the sets Σ, Γ and q. In the worst case it may therefore be exponential in $|\Gamma|$ which is not avoidable in view of Lemma 4. To compute $\mathsf{ReachOrSafe}(\mathcal{T}, \mathcal{F})$, we evaluate the following fixpoint formula in the lattice of antichains: $\varphi_1 \equiv \nu X.(\mathcal{F} \sqcap \mathsf{CPre}(X)) \sqcup \mathcal{T}^*$ where $\mathcal{T}^* = \mu X.\mathsf{CPre}(X) \sqcup \mathcal{T}$. To compute $\mathsf{ReachAndSafe}(\mathcal{T}, \mathcal{F})$, we use $\varphi_2 \equiv \mu X.\mathcal{F} \sqcap (\mathsf{CPre}(X) \sqcup \mathcal{T})$.

When computing $q' = \mathsf{CPre}(q)$, we associate with each cell in the antichain q' the action to be played in order to ensure reaching a set in q. For φ_1, this information is sufficient to extract a winning strategy from the fixpoint: the action associated with each winning cell ensures to reach an element of the fixpoint, thus either confining the game inside $\mathcal{F}$ forever, or eventually reaching $\mathcal{T}^*$. On the other hand, for $\mathcal{T}^*$ and φ_2 (which has the flavor of reachability), we have seen in Section 3 that the final fixpoint is not sufficient to recover the winning strategy. Therefore, we have to construct on the fly the winning strategy while computing the fixpoint. We output a reachability strategy as a tree structure whose nodes are the sets in the successive antichains computed in the least-fixpoint iterations together with their associated action $\sigma \in \Sigma$. If $q' = \mathsf{CPre}(q)$ and σ is the action to be played in cell $s \in q'$, then for each observation o (given by Player 2) we know that there exists a cell $s_o \in q$ such that $\mathsf{post}(s) \cap \gamma(o) \subseteq s_o$. Correspondingly, each node for the sets in q' has $|\Gamma|$ outgoing edges to some sets in q.To evaluate the scalability of our algorithm, we have generated game structures and objectives randomly. We fixed the alphabet $\Sigma = \{0, 1\}$ and we used

the following parameters to generate game instances: the *size* $|L|$ of the game, the *transition density* $r = \frac{|\Delta|}{|L|\cdot|\Sigma|}$, i.e., the average branching degree of the game graph, and the *density* $f = \frac{|\Gamma|}{|L|}$ *of observations*. For each $\sigma \in \Sigma$, we generate $r \cdot |L|$ pairs $(\ell, \ell') \in L \times L$ uniformly at random; each location is randomly assigned one of the $f \cdot |L|$ observations. We have tested reachability and Büchi objectives for games with transition density varying from 0.5 to 4 and density of observation varying from 0.1 to 0.9. We have limited the execution time to 10s for each instance. The size of the generated instances ranges from 50 to 500. For all values of the parameters, our prototype solved half of the instances of size 100 for both reachability and Büchi objectives. When the transition density is below 1.5, the instances are much easier to solve and the maximal size is 350 for reachability and 200 for Büchi objectives. Finally, we did not observe significant influence of the number of observations on the performance of the prototype. It seems that the exponential cost of computing $\mathsf{CPre}(\cdot)$ is compensated by the fact that for large number of observations, the games are closer to perfect-information games.

References

1. Berwanger, D., Chatterjee, K., Doyen, L., Henzinger, T.A., Raje, S.: Strategy construction for parity games with imperfect information. Technical Report MTC-REPORT-2008-005, EPFL (2008), http://infoscience.epfl.ch/record/125011
2. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for omega-regular games of incomplete information. Logical Methods in Computer Science 3(3:4) (2007)
3. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
4. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.-F.: Antichains: Alternative algorithms for LTL satisfiability and model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 63–77. Springer, Heidelberg (2008)
5. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: Proc. of FoCS 1991, pp. 368–377. IEEE, Los Alamitos (1991)
6. Fudenberg, D., Tirole, J.: Game Theory. MIT Press, Cambridge (1991)
7. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
8. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 886–902. Springer, Heidelberg (2003)
9. McNaughton, R.: Infinite games played on finite graphs. Annals of Pure and Applied Logic 65(2), 149–184 (1993)
10. Reif, J.: The complexity of two-player games of incomplete information. Journal of Computer and System Sciences 29, 274–301 (1984)
11. Thomas, W.: On the synthesis of strategies in infinite games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
12. Thomas, W.: Languages, automata, and logic. Handbook of Formal Languages 3, 389–455 (1997)
13. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science 200, 135–183 (1998)

Mixing Lossy and Perfect Fifo Channels*
(Extended Abstract)

P. Chambart and Ph. Schnoebelen

LSV, ENS Cachan, CNRS
61, av. Pdt. Wilson, F-94230 Cachan, France
{chambart,phs}@lsv.ens-cachan.fr

Abstract. We consider asynchronous networks of finite-state systems communicating via a combination of reliable and lossy fifo channels. Depending on the topology, the reachability problem for such networks may be decidable. We provide a complete classification of network topologies according to whether they lead to a decidable reachability problem. Furthermore, this classification can be decided in polynomial-time.

1 Introduction

Fifo channels. Channel systems, aka "communicating finite-state machines", are a classical model for protocols where components communicate asynchronously via fifo channels [8]. When the fifo channels are unbounded, the model is Turing-powerful since channels can easily be used to simulate the tape of a Turing machine.

It came as quite a surprise when Abdulla and Jonsson [3, 4], and independently Finkel *et al.* [13], showed that *lossy* channel systems (LCS's), i.e., channel systems where one assumes that the channels are unreliable so that messages can be lost nondeterministically, are amenable to algorithmic verification (see also [20]). The model has since been extended in several directions: message losses obeying probability laws [1, 2, 6, 21], channels with other kinds of unreliability [7, 9], etc.

How this unreliability leads to decidability is paradoxical, and hard to explain in high-level, non-technical terms. It certainly does not make the model trivial: we recently proved that LCS verification is exactly at level $\mathcal{F}_{\omega^\omega}$ in the extended Grzegorczyk Hierarchy, hence it is not primitive-recursive, or even multiply-recursive [12].

An ubiquitous model. In recent years, lossy channels have shown up in unexpected places. They have been used in reductions showing hardness (or less frequently decidability) for apparently unrelated problems in modal logics [16], in temporal logics [19], in timed automata [17], in data-extended models [15], etc. More and more, LCS's appear to be a pivotal model whose range goes far beyond asynchronous protocols.

* Work supported by the Agence Nationale de la Recherche, grant ANR-06-SETIN-001.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 340–355, 2008.

Fueling this line of investigation, we recently discovered that the "Regular Post Embedding Problem", a new decidable variant of Post's Correspondence Problem, is equivalent (in a non-trivial way) to LCS reachability [10, 11]. This discovery was instigated by a study of *unidirectional* channel systems (UCS), where a Sender can send messages to a Receiver via two fifo channels, one reliable and one lossy, but where there is no communication in the other direction (see also topology T_2^d in Fig. 1 below). As far as we know, this simple arrangement had never been studied before.

Our contribution. This paper considers the general case of *mixed* channel systems, where some channels are reliable and some are lossy. These systems can be Turing-powerful (one process using one reliable fifo buffer is enough) but not all network topologies allow this (e.g., systems with only lossy channels, or systems where communication is arranged in a tree pattern with no feedback, or UCS's as above). Our contribution is a complete classification of network topologies according to whether they lead to undecidable reachability problems, or not. This relies on original and non-trivial transformation techniques for reducing large topologies to smaller ones while preserving decidability.

Beyond providing a complete classification, the present contribution has several interesting outcomes. First, we discovered new decidable configurations of channel systems, as well as new undecidable ones, and these new results are often surprising. They enlarge the existing toolkit currently used when transferring results from channel systems to other areas, according to the "ubiquitous model" slogan. Secondly, the transformation techniques we develop may eventually prove useful for reducing/delaying the combinatorial explosion one faces when verifying asynchronous protocols.

Outline of the paper. We describe *mixed channel systems* and their topologies in Section 2 and provide in Section 3 a few original results classifying the basic topologies to which we reduce larger networks. Section 4 shows that "fusing essential channels" preserves decidability. An additional "splitting" technique is described in Section 5. After these three sections, we have enough technical tools at hand to describe our main result, the complete classification method, and prove its correctness in Sections 6 and 7.

2 Systems with Reliable and Lossy Channels

We classify channel systems according to their *network topology*, which is a graph describing who are the participant processes and what channels they are connected to.

2.1 Network Topologies

Formally, a *network topology*, or shortly a *topology*, is a tuple $T = \langle N, R, L, s, d \rangle$ where N, R and L are three mutually disjoint finite sets of, respectively, *nodes*,

reliable channels, and *lossy channels*, and where, writing $C \stackrel{\text{def}}{=} R \cup L$ for the set of channels, $s, d : C \to N$ are two mappings that associate a *source* and a *destination* node to each channel. For a node $n \in N$, $out(n) = \{c \in C \mid s(c) = n\}$ is the set of output channels for n, while $in(n) = \{c \in C \mid d(c) = n\}$ is the set of its input channels.

Graphical examples of simple topologies can be found below, where we use dashed arrows to single out the lossy channels (reliable channels are depicted with full arrows). We do not distinguish between isomorphic topologies since N, R and L simply contain "names" for nodes and channels: these are irrelevant here and only the directed graph structure with two types of edges matters.

2.2 Mixed Channel Systems and Their Operational Semantics

Assume $T = \langle N, R, L, s, d\rangle$ is a topology with n nodes, i.e., with $N = \{P_1, P_2, ..., P_n\}$. Write $C = R \cup L$ for the set of channels. A *mixed channel system* (MCS) having topology T is a tuple $S = \langle T, \mathtt{M}, Q_1, \Delta_1, ..., Q_n, \Delta_n\rangle$ where $\mathtt{M} = \{\mathtt{a}, \mathtt{b}, ...\}$ is a finite *message alphabet* and where, for $i = 1, ..., n$, Q_i is the finite set of (control) states of a process (also denoted P_i) that will be located at node $P_i \in N$, and Δ_i is the finite set of *transition rules*, or shortly "rules", governing the behaviour of P_i. A rule $\delta \in \Delta_i$ is either a *writing rule* of the form $(q, c, !, \mathtt{a}, q')$ with $q, q' \in Q_i$, $c \in out(P_i)$ and $\mathtt{a} \in \mathtt{M}$ (usually denoted "$q \xrightarrow{c!\mathtt{a}} q'$"), or it is a *reading rule* $(q, c, ?, \mathtt{a}, q')$ (usually denoted "$q \xrightarrow{c?\mathtt{a}} q'$") with this time $c \in in(P_i)$. Hence the way a topology T is respected by a channel system is via restrictions upon the set of channels to which a given participant may read from, or write to.

Our terminology "*mixed* channel system" is meant to emphasize the fact that we allow systems where lossy channels coexist with reliable channels.

The behaviour of some $S = \langle T, \mathtt{M}, Q_1, \Delta_1, ..., Q_n, \Delta_n\rangle$ is given under the form of a transition system. Assume $C = \{c_1, ..., c_k\}$ contains k channels. A configuration of S is a tuple $\sigma = \langle q_1, ..., q_n, u_i, ..., u_k\rangle$ where, for $i = 1, ..., n$, $q_i \in Q_i$ is the current state of P_i, and where, for $i = 1, ..., k$, $u_i \in \mathtt{M}^*$ is the current contents of channel c_i.

Assume $\sigma = \langle q_1, ..., q_n, u_i, ..., u_k\rangle$ and $\sigma' = \langle q'_1, ..., q'_n, u'_i, ..., u'_k\rangle$ are two configurations of some system S as above, and $\delta \in \Delta_i$ is a rule of participant P_i. Then δ witnesses a transition between σ and σ', also called a *step*, and denoted $\sigma \xrightarrow{\delta} \sigma'$, if and only if

- the control states agree with, and are modified according to δ, i.e., $q_i = q$, $q'_i = q'$, $q_j = q'_j$ for all $j \neq i$;
- the channel contents agree with, and are modified according to δ, i.e., either
  - $\delta = (q, c_l, ?, \mathtt{a}, q')$ is a reading rule, and $u_l = \mathtt{a}.u'_l$, or
  - $\delta = (q, c_l, !, \mathtt{a}, q')$ is a writing rule, and $u'_l = u_l.\mathtt{a}$, or $c_l \in L$ is a lossy channel and $u'_l = u_l$;

  in both cases, the other channels are untouched: $u'_j = u_j$ for all $j \neq l$.

Such a step is called "*a step by* P_i" and we say that its *effect* is "reading $\mathtt{a}$ on c", or "writing $\mathtt{a}$ to c", or "losing $\mathtt{a}$". A *run* (from σ_0 to σ_n) is a sequence of steps of the form $r = \sigma_0 \xrightarrow{\delta_1} \sigma_1 \xrightarrow{\delta_2} \sigma_2 \cdots \xrightarrow{\delta_n} \sigma_n$. A run is *perfect* if none of its steps loses a message.

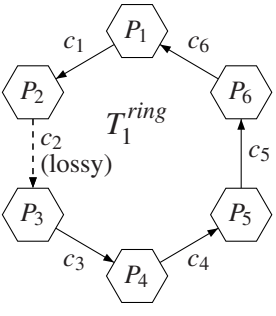
Remark 2.1. With this operational semantics for lossy channels, messages can only be lost when a rule writes them to a channel. Once inside the channels, messages can only be removed by reading rules. This definition is called the *write-lossy* semantics for lossy channels: it differs from the more classical definition where messages in lossy channels can be lost at any time. We use it because it is the most convenient one for our current concerns, and because this choice does not impact the reachability questions we consider (see [12, Appendix A] for a formal comparison). □

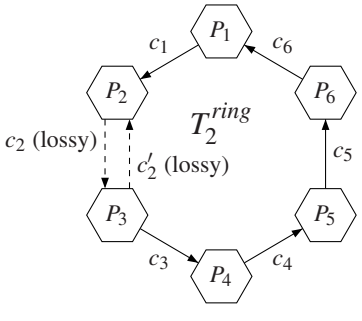
2.3 The Reachability Problem for Network Topologies

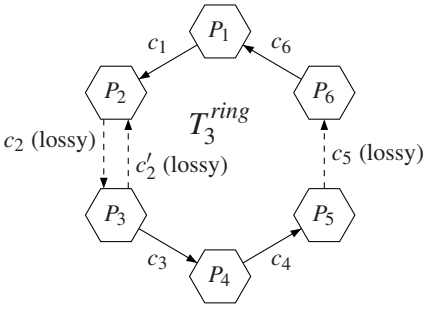
The *reachability problem* for mixed channel systems asks, for a given S and two configurations $\sigma_{\text{init}} = \langle q_1, \ldots, q_n, \varepsilon, \ldots, \varepsilon \rangle$ and $\sigma_{\text{final}} = \langle q'_1, \ldots, q'_n, \varepsilon, \ldots, \varepsilon \rangle$ in which the channels are empty, whether S has a run from σ_{init} to σ_{final}. That we restrict reachability questions to configurations with empty channels is technically convenient, but it is no real loss of generality.

The *reachability problem* for a topology T is the restriction of the reachability problem to mixed systems having topology T. Hence if reachability is decidable for T, it is decidable for all MCS's having topology T. If reachability is not decidable for T, it may be decidable or not for MCS's having topology T (but it must be undecidable for one of them). Finally, if T' is a subgraph of T and reachability is decidable for T, then it is for T' too.

Our goal is to determine for which topologies reachability is decidable. Let us illustrate the questions and outline some of our results. T_1^{ring} is a topology describing a directed ring of processes, where each participant sends to its right-hand neighbour, and receives from its left-hand neighbour. One of the channels is lossy. A folk claim is that such cyclic networks have decidable reachability as soon as one channel is lossy. The proof ideas behind this claim have not been formally published and they do not easily adapt to related questions like "what about T_2^{ring}?", where a lossy channel in the other direction is added, or about T_3^{ring} where we start with more lossy channels in the ring.







Our techniques answer all three questions uniformly. One of our results states that all channels along the path c_3 to c_4 to c_5 to c_6 to c_1 can be fused into a single channel going from P_3 to P_2 without affecting the decidability of reachability. The transformations are modular (we fuse one channel at a time). Depending on the starting topology, we end up with different two-node topologies, from which we deduce that T_1^{ring} and T_3^{ring} have decidable reachability, while T_2^{ring} does not (see Corollary 4.6 below).

3 Reachability for Basic Topologies

This section is concerned with the basic topologies to which we will later reduce all larger cases.

Theorem 3.1 (Basic topologies). *Reachability is decidable for the network topologies T_1^d and T_2^d (see Fig. 1). It is not decidable for the topologies T_1^u, T_2^u, T_3^u, T_4^u, T_5^u, and T_6^u (see Fig. 2).*

We start with the decidable cases:

That T_1^d, and more generally all topologies with only lossy channels (aka LCS's), leads to decidable problems is the classic result from [4].

Regarding T_2^d, we recently proved it has decidable reachability in [10], where T_2^d-systems are called "unidirectional channel systems", or UCS's. Our reason for investigating UCS's was indeed that this appeared as a necessary preparation for the classification of mixed topologies. Showing that T_2^d has decidable reachability is quite involved, going through the introduction of the "Regular Post Embedding Problem". In addition, [10, 11] exhibit non-trivial reductions between reachability for UCS's and reachability for LCS's: the two problems are equivalent.

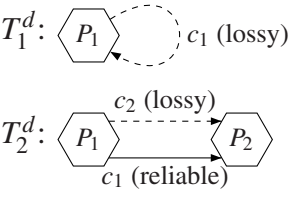


Fig. 1. Basic decidable topologies

Now to the undecidable cases:

It is well-known that T_1^u may lead to undecidable problems [8], and this is also known, though less well, for T_2^u (restated, e.g., as the non-emptiness problem for the intersection of two rational transductions). The other four results mix lossy and reliable channels and are new. We actually prove all six cases in a uniform framework, by reduction from Post's Correspondence Problem, aka PCP, or its directed variant, $\mathsf{PCP}_{\mathrm{dir}}$.

Recall that an instance of PCP is a family $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ of $2n$ words over some alphabet. The question is whether there is a non-empty sequence (a *solution*) $i_1, \ldots, i_k$ of indexes such that $x_{i_1} x_{i_2} \ldots x_{i_k} = y_{i_1} y_{i_2} \ldots y_{i_k}$. $\mathsf{PCP}_{\mathrm{dir}}$ asks whether there is a *directed* solution $i_1, \ldots, i_k$, i.e., a solution such that, in

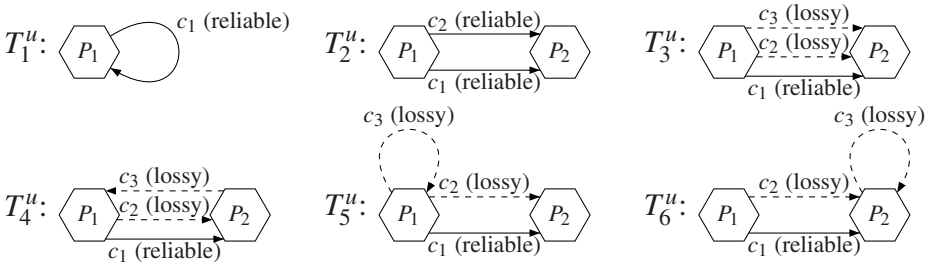


Fig. 2. Basic topologies with undecidable reachability

addition, $y_{i_1}y_{i_2}\dots y_{i_h}$ is a prefix of $x_{i_1}x_{i_2}\dots x_{i_h}$ for all $h = 1,\dots,k$. It is well-known that PCP and $\mathsf{PCP}_{\text{dir}}$ are undecidable, and more precisely Σ_0^1-complete.

Reducing PCP *to* T_2^u*-networks.* With a PCP instance $(x_i, y_i)_{i=1,\dots,n}$, we associate a process P_1 having a single state p_1 and n loops[1] $p_1 \xrightarrow{c_1!x_i \;\; c_2!y_i} p_1$, one for each index $i = 1, ..., n$. Process P_1 guesses a solution $i_1 i_2 i_3 \dots$ and sends the concatenations $x_{i_1}x_{i_2}x_{i_3}\dots$ and $y_{i_1}y_{i_2}y_{i_3}\dots$ on, respectively, c_1 and c_2. Process P_2 checks that the two channels c_1 and c_2 have the same contents, using reading loops $p_2 \xrightarrow{c_1?\mathsf{a} \;\; c_2?\mathsf{a}} p_2$, one for each symbol $\mathsf{a}, \mathsf{b}, \dots$ in the alphabet. An extra control state, for example p_1' with rules $p_1' \xrightarrow{c_1!x_i \;\; c_2!y_i} p_1$, is required to check that P_1 picks a non-empty solution. Then, in the resulting T_2^u-network, $\langle p_1', p_2, \varepsilon, \varepsilon\rangle \xrightarrow{*} \langle p_1, p_2, \varepsilon, \varepsilon\rangle$ if and only if the PCP instance has a solution.

Reducing PCP *to* T_3^u*-networks.* For T_3^u, the same idea is adapted to a situation with three channels, two of which are lossy. Here P_1 has rules $p_1 \xrightarrow{c_2!x_i \;\; c_3!y_i \;\; c_1!1^{|x_i y_i|}} p_1$. Thus P_1 sends x_i and y_i on lossy channels and simultaneously sends the number of letters in unary (1 is a special tally symbol) on c_1, the perfect channel. P_2 matches these with reading loops of the form $p_2 \xrightarrow{c_1?1\mathsf{1} \;\; c_2?\mathsf{a} \;\; c_3?\mathsf{a}} p_2$ for each letter a. If P_2 can consume all 1's out of c_1, this means that no message has been lost on the lossy channels, and then P_2 really witnessed a solution to the PCP instance.

Reducing PCP_{dir} *to* T_1^u*-networks.* For T_1^u, we consider the directed $\mathsf{PCP}_{\text{dir}}$. P_1 has n loops $p_1 \xrightarrow{c_1!x_i \;\; c_1?y_i} p_1$ where the guessing and the matching is done by a single process. Since at any step $h = 1, ..., k$ the concatenation $x_{i_1}x_{i_2}...x_{i_h}$ is (partly) consumed while matching for $y_{i_1}y_{i_2}...y_{i_h}$, only directed solutions will be accepted.

[1] Transition rules like "$p_1 \xrightarrow{c_1!x_i \;\; c_2!y_i} p_1$" above, where several reads and writes are combined in a same rule, and where one writes or reads words rather than just one message at a time, are standard short-hand notations for sequences of rules using intermediary states that are left implicit. We avoid using this notation in situations where the specific ordering of the combined actions is important as, e.g., in (*) below.

Reducing PCP_{dir} *to* T_5^u*-networks.* For T_5^u too, we start from $\mathsf{PCP}_{\mathrm{dir}}$ and use a variant of the previous counting mechanism to detect whether some messages have been lost. P_1 has rules of the form $p_1 \xrightarrow{c_3!1^{|x_i|}\ c_1!x_i\ c_3?1^{|y_i|}\ c_2!y_i} p_1$, i.e., it sends x_i on c_1 (the reliable channel) and y_i on c_2 (unreliable) while P_2 checks the match with loops $p_2 \xrightarrow{c_1?\mathsf{a}\ \ c_2?\mathsf{a}} p_2$. In addition, P_1 also maintains in c_3 a count of the number of symbols written to c_1 minus the number of symbols written to c_2, or $\#_h \stackrel{\text{def}}{=} |x_{i_1} \dots x_{i_h}| - |y_{i_1} \dots y_{i_h}|$. The counting scheme forbids partial sequences $y_{i_1} \dots y_{i_h}$ that would be longer than the corresponding $x_{i_1} \dots x_{i_h}$, but this is right since we look for directed solutions. If tally symbols on c_3 are lost, or if part of the y_i's on c_2 are lost, then it will never be possible for P_2 to consume all messages from c_1. Finally a run from $\langle p_1', p_2, \varepsilon, \varepsilon, \varepsilon\rangle$ to $\langle p_1, p_2, \varepsilon, \varepsilon, \varepsilon\rangle$ must be perfect and witnesses a directed solution.

Reducing PCP_{dir} *to* T_6^u*-networks.* For T_6^u, we adapt the same idea, this time having P_2 monitoring the count $\#_h$ on c_3. P_1 has loops $p_1 \xrightarrow{c_1!x_i 1^{|y_i|}\ \ c_2!y_i 1^{|x_i|}} p_1$ where a guessed solution is sent on c_1 and c_2 with interspersed tally symbols. The guessed solution is checked with the usual loops $p_2 \xrightarrow{c_1?\mathsf{a}\ \ c_2?\mathsf{a}} p_2$. The 1's on c_2 are stored to c_3 and matched (later) with the 1's on c_1 via two loops: $p_2 \xrightarrow{c_2?1\ \ c_3!1} p_2$ and $p_2 \xrightarrow{c_3?1\ \ c_1?1} p_2$. In a reliable run, there is always as many messages on c_1 as there are on c_2 and c_3 together, and strictly more if a message is lost. Hence a run from $\langle p_1', p_2, \varepsilon, \varepsilon, \varepsilon\rangle$ to $\langle p_1, p_2, \varepsilon, \varepsilon, \varepsilon\rangle$ must be perfect and witness a solution. Only direct solutions can be accepted since the tally symbols in c_3 count $\#_h$ that cannot be negative.

Reducing PCP_{dir} *to* T_4^u*-networks.* For T_4^u, we further adapt the idea, again with the count $\#_h$ stored on c_3 but now sent from P_2 to P_1. The loops in P_1 now are

$$p_1 \xrightarrow{c_1!x_i\ \ c_2!y_i 1^{|x_i|}} q_i \xrightarrow{c_3?1^{|y_i|}} p_1. \qquad (*)$$

The 1's on c_2 are sent back via c_3 to be matched later by P_1, thanks to a loop $p_2 \xrightarrow{c_2?1\ \ c_3!1} p_2$. Again a message loss will leave strictly more messages in c_1 than in c_2 and c_3 together, and cannot be recovered from. Only direct solutions can be accepted since the tally symbols in c_3 count $\#_h$.

4 Fusion for Essential Channels

This section and the following develop techniques for "simplifying" topologies while preserving the decidability status of reachability problems.

We start with a reduction called "fusion".

Let $T = \langle N, R, L, s, d\rangle$ be a network topology. For any channel $c \in C$, $T - c$ denotes the topology obtained from T by deleting c. For any two distinct nodes $P_1, P_2 \in N$, $T[P_1 = P_2]$ denotes the topology obtained from T by merging P_1 and P_2 in the obvious way: channel extremities are redirected accordingly.

Clearly, any MCS with topology $T - c$ can be seen as having topology T. Thus $T - c$ has decidable reachability when T has, but the converse is not true in general.

Similarly, any MCS having topology T can be transformed into an equivalent MCS having topology $T[P_1 = P_2]$ (using the asynchronous product of two control automata). Thus T has decidable reachability when $T[P_1 = P_2]$ has, but the converse is not true in general.

For any channel c such that $s(c) \neq d(c)$, we let T/c denote $T[s(c) = d(c)] - c$ and say that T/c is "*obtained from T by contracting c*". Hence T/c is obtained by merging c's source and destination, and then removing c.

Since T/c is obtained via a combination of merging and channel removal, there is, in general, no connection between the decidability of reachability for T and for T/c. However, there is a strong connection for so-called "essential" channels, as stated in Theorem 4.5 below.

Before we can get to that point, we need to explain what is an essential channel and how they can be used.

4.1 Essential Channels Are Existentially 1-Bounded

In this section, we assume a given MCS $S = \langle T, \mathtt{M}, Q_1, \Delta_1, ..., Q_n, \Delta_n \rangle$ is a MCS with $T = \langle N, R, L, s, d \rangle$.

Definition 4.1. *A channel $c \in C$ is* essential *if $s(c) \neq d(c)$ and all directed paths from $s(c)$ to $d(c)$ in T go through c.*

In other words, removing c modifies the connectivity of the directed graph underlying T.

The crucial feature of an essential channel c is that causality between the actions of $s(c)$ and the actions of $d(c)$ is constrained. As a consequence, it is always possible to reorder the actions in a run so that reading from c occurs immediately after the corresponding writing to c. As a consequence, bounding the number of messages that can be stored in c does not really restrict the system behaviour.

Formally, for $b \in \mathbb{N}$, we say a channel c is b-bounded along a run $\pi = \sigma_0 \xrightarrow{\delta_1} \dots \xrightarrow{\delta_n} \sigma_n$ if $|\sigma_i(c)| \leq b$ for $i = 0, \dots, n$. We say c is *synchronous* in π if it is 1-bounded and at least of $\sigma_i(c)$ and $\sigma_{i+1}(c) = \varepsilon$ for all $0 \leq i < n$. Hence a synchronous channel only stores at most one message at a time, and the message is read immediately after it has been written to c.

Proposition 4.2. *If c is essential and $\pi = \sigma_0 \xrightarrow{\delta_1} \dots \xrightarrow{\delta_n} \sigma_n$ is a run with $\sigma_0(c) = \sigma_n(c) = \varepsilon$, then S has a run π' from σ_0 to σ_n in which c is synchronous.*

This notion is similar to the existentially-bounded systems of [18] but is applies to a single channel, not to the whole system.

We prove Proposition 4.2 using techniques and concepts from true concurrency theory and message flow graphs (see, e.g., [14]). With a run $\pi = \sigma_0 \xrightarrow{\delta_1} \dots \xrightarrow{\delta_n} \sigma_n$ as above, we associate a set $E = \{1, \dots, n\}$ of n events, that can be thought of the

actions performed by the n steps of π: firing a transition and reading or writing or losing a message. Observe that different occurrences of a same transition with same effect are two different events. We simply identify the events with indexes from 1 to n. We write $e, e', \ldots$ to denote events, and also use r and w for reading and writing events.

Any $e \in E$ is an event of some process $N(e) \in N$ and we write $E = \bigcup_{P \in N} E_P$ the corresponding partition. There exist several (standard) causality relations between events. For every process $P \in n$, the events of P are linearly ordered by $<_P$: $i <_P j$ iff $i, j \in E_P$ and $i < j$. For every channel $c \in C$, the events that write to or read from c are related by $<_c$ with $i <_c j$ iff i is an event that writes some m to c, and j is the event that reads that (occurrence of) m. (Here, events that lose messages are considered as internal actions where no channel is involved.) We let $\prec$ (and $\preccurlyeq$) denote the transitive (resp. reflexive-transitive) closure of $\bigcup_{P \in N} <_P \cup \bigcup_{c \in C} <_c$. $(E, \preccurlyeq)$ is then a poset, and $\preccurlyeq$ is called the *visual* order (also causality order, or dependency order) in the literature. For $e \in E$, we let $\uparrow e$ denote the past of e, i.e., the set $\{e' \in E \mid e' \preccurlyeq e\}$.

It is well-known that any linear extension $e_1, \ldots, e_n$ of $(E, \preccurlyeq)$ is causally consistent and can be transformed into a run $\pi' = \sigma_0 \xrightarrow{e_1} \xrightarrow{e_2} \cdots$ starting from σ_0. This run ends in σ_n like π, though it may go through different intermediary configurations. All the runs obtained by considering different linear extensions are causally equivalent to π, denoted $\pi \approx \pi'$, and they all give rise to the same poset $(E, \preccurlyeq)$.

We now state properties enjoyed by $(E, \preccurlyeq)$ in our context that are useful for proving Proposition 4.2. First, observe that, since the channels are fifo, and since only one process, namely $d(c)$ (resp. $s(c)$), is allowed to read from (resp. write to) a channel c:

$$(w_1 <_c r_1 \text{ and } w_2 <_c r_2) \text{ imply } (w_1 <_{s(c)} w_2 \text{ iff } r_1 <_{d(c)} r_2). \qquad (\dagger)$$

Another important observation is the following: assume $e \preccurlyeq e'$. Then, and since $\preccurlyeq$ is defined as a reflexive-transitive closure, there must be a chain of the form

$$e = e_0 \leq_{P_0} e'_0 <_{c_1} e_1 \leq_{P_1} e'_1 <_{c_2} \cdots <_{c_l} e_l \leq_{P_l} e'_l = e'$$

where, for $1 \leq i \leq l$, $s(c_i) = P_{i-1}$ and $d(c_i) = P_i$. Hence T has a path $c_1, \ldots, c_l$ going from P_0 to P_l.

Lemma 4.3. *If $e_1 \prec e_2 \prec e_3$ and c is essential, then $e_1 \not<_c e_3$.*

Proof. By contradiction. Assume $e_1 \prec e_2 \prec e_3$ and $e_1 <_c e_3$ for an essential c. Then $e_1 \in E_P$, $e_3 \in E_{P'}$ and, since all paths from P to P' go through c (by essentiality of c), there must exist a pair $w, r \in E$ with $e_1 \preccurlyeq w <_c r \preccurlyeq e_2$ or, symmetrically, $e_2 \preccurlyeq w <_c r \preccurlyeq e_3$, depending on whether the $w <_c r$ pair occurs before or after e_2 in the chain from e_1 to e_2 to e_3. If $e_1 \preccurlyeq w <_c r \preccurlyeq e_2 \prec e_3$, then $r <_{P'} e_3$, hence $w <_P e_1$ using (†). If $e_1 \prec e_2 \preccurlyeq w <_c r \preccurlyeq e_3$, then $e_1 <_P w$, hence $e_3 <_{P'} r$ using (†). In both cases we obtain a contradiction. □

We now assume that c is essential and that π has $\sigma_0(c) = \sigma_n(c) = \varepsilon$ (hence E has the same number, say m, of events reading from c and writing to it). Write P for $s(c)$ and P' for $d(c)$. Let $w_1 <_P w_2 \ldots <_P w_m$ be the m events that write to c, listed in causal order. Let $r_1 <_{P'} e_2 \ldots <_{P'} r_m$ be the m events that read from c listed in causal order.

Lemma 4.4. *There exists a linear extension of $(E, \preccurlyeq)$ where, for $i = 1, \ldots, m$, w_i occurs just before r_i.*

Proof. The linear extension is constructed incrementally. Formally, for $i = 1, \ldots, m$, let $E_i \stackrel{\text{def}}{=} {\uparrow} r_i$ and $E'_i \stackrel{\text{def}}{=} E_i \setminus \{r_i, w_i\}$. Observe that $F_1 \subset E_1 \subseteq F_2 \cdots F_i \subset E_i \subseteq F_{i+1}$, with the convention that $F_{m+1} = E$. Every E_i is a $\preccurlyeq$-closed subset of E, also called a down-cut of $(E, \preccurlyeq)$. Furthermore, F_i is a down-cut of E_i by Lemma 4.3. Hence a linear extension of F_i followed by $w_i.r_i$ gives a linear extension of E_i, and following it with a linear extension of $F_{i+1} \setminus E_i$ gives a linear extension of F_{i+1}. Any linear extension of $F_{i+1} \setminus E_i$ can be chosen since this subset does not contain reads from, or writes to c. □

The linear extension we just built gives rise to a run π' in which c is synchronous, which concludes the proof of Proposition 4.2.

Observe that when several channels are essential in T, it is in general not possible to replace a run π with an equivalent π' where all essential channels are simultaneously synchronous.

4.2 Decidability by Fusion

We call "*fusion*" the transformation of T to T/c where c is essential, and "*reliable fusion*" the special case where c is also a reliable channel.

Theorem 4.5 (Decidability by fusion). *Let c be an essential channel in T:*

1. T has decidable reachability when T/c has.
2. If c is a reliable channel, then T/c has decidable reachability when T has.

Proof. 1. Let S be a T-MCS. We replace it by a system S' where c has been removed and where the processes at nodes $P_1 = s(c)$ and $P_2 = d(c)$ have been replaced by a larger process that simulate both P_1 and P_2 and where communication along c is replaced by synchronizing the sends in P_1 with the reads in P_2 (message losses are simulated even more simply by the P_1 part). S' has topology T/c and simulates S *restricted to runs where c is synchronous*. By Proposition 4.2, this is sufficient to reach any reachable configuration. Since reachability in S' is decidable, we conclude that reachability in S is decidable.

2. We now also assume that c is reliable and consider a (T/c)-MCS S. With S we associate a T-MCS S' that simulates S. S' has two nodes P_1 and P_2 where S only had a merged P node.

The construction is illustrated in Fig. 3. Informally, P_1 inherits states from P and all rules that read from channels c_1 with $d(c_1) = P_1$ in T, or write to channels c_2 with $s(c_2) = P_1$. Regarding the other rules, the communication

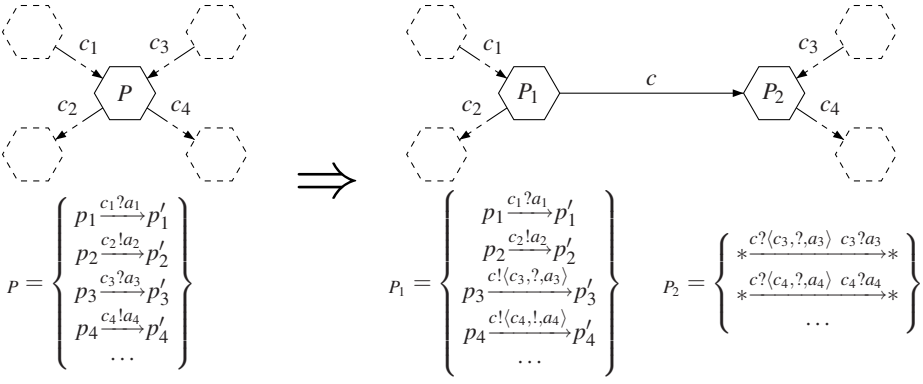


Fig. 3. Associating a T-MCS with a T/c-MCS

action (reading from some c_3 or writing to some c_4) is sent to P_2 via c. S' uses an extended alphabet $\mathtt{M}'$ that extends the message alphabet $\mathtt{M}$ from S via $\mathtt{M}' \stackrel{\text{def}}{=} \mathtt{M} \cup (C \times \{?,!\} \times \mathtt{M})$. P_2 only has simple loops around a central state $*$ that read communication instructions from P_1 via c and carry them out.

S' simulates S in a strong way. Any step in S can be simulated in S', perhaps by two consecutive steps if a communication operation has to transit from P_1 to P_2 via c. In the other direction, there are some runs in S' that cannot be simulated directly by S, e.g., when P_2 does not carry out the instructions sent by P_1 (or carries them out with a delay). But all runs in S' in which c is synchronous are simulated by S.

Since runs in which c is synchronous are sufficient to reach any configuration reachable in S' (Proposition 4.2), the two-way simulation reduces reachability in S to reachability in S', which is decidable if T has decidable reachability. □

The usefulness of Theorem 4.5 is illustrated by the following two corollaries.

Corollary 4.6. *T_1^{ring} and T_3^{ring} (from Section 2.1) have decidable reachability. T_2^{ring} does not.*

Proof. Building $T_1^{ring}/c_3/c_4/c_5/c_6/c_1$ only fuses essential channels and ends up with a decidable topology (only lossy channels).

Starting with T_2^{ring}, we can build $T = T_2^{ring}/c_3/c_4/c_5/c_6$ but have to stop there since c_1 is not essential in the resulting T: there now is another path using c_2'. The resulting T, isomorphic to T_4^u from Fig. 2, does not have decidable reachability. Hence T_2^{ring} does not have decidable reachability since we fused reliable channels only.

With T_3^{ring}, it is better to build $T_3^{ring}/c_3/c_4/c_6/c_1$. Here too we cannot fuse any more because of c_2', but the end result is a topology with decidable reachability since c_5 is lossy. Hence T_3^{ring} has decidable reachability. □

Corollary 4.7. *A topology in the form of a forest has decidable reachability.*

Proof (Sketch). If T is a forest, every channel c is essential, and every T/c is still a forest. Hence T reduces to a topology with no channels (i.e., a collection of disconnected nodes) where reachability is clear. □

5 Splitting Along Lossy Channels

Let $T_1 = \langle N_1, R_1, L_1, s_1, d_1 \rangle$ and $T_2 = \langle N_2, R_2, L_2, s_2, d_2 \rangle$ be two disjoint topologies. We say that $T = \langle N, R, L, s, d \rangle$ is a *(lossy) gluing of* T_1 *on* T_2 if T is a juxtaposition of T_1 and T_2 (hence $N = N_1 \cup N_2$, ...) with an additional set L_3 of lossy channels (hence $R = R_1 \cup R_2$ and $L = L_1 \cup L_2 \cup L_3$) connecting from T_1 to T_2 in a unidirectional way: $s(L_3) \subseteq N_1$ and $d(L_3) \subseteq N_2$.

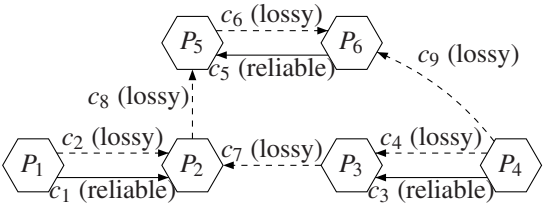


Fig. 4. A topology that splits in three

This situation is written informally "$T = T_1 \triangleright T_2$", omitting details on L_3 and its connections. In practice this notion is used to split a large T into subparts rather than build larger topologies out of T_1 and T_2.

Theorem 5.1 (Decidability by splitting). *Reachability is decidable for* $T_1 \triangleright T2$ *if, and only if, it is for both* T_1 *and* T_2.

The proof of Theorem 5.1 (omitted here, see full version of this paper) uses techniques that are standard for LCS's but that have to be adapted to the more general setting of MCS's.

We can apply Theorem 5.1 to prove that the topology in Fig. 4 has decidable reachability. Indeed, this topology can be split along lossy channels, namely along c_7, c_8 and c_9, giving rise to two copies of T_2^d (from Fig. 1) and a two-node ring that can be reduced to T_1^d by fusion.

6 A Complete Classification

In this section, we prove that the results from the previous sections provide a complete classification.

Theorem 6.1 (Completeness). *A network topology* T *has decidable reachability if, and only if, it can be reduced to* T_2^d *(from Fig. 1) and LCS's using fusion and splitting only.*[2]

[2] As is well-known, it is possible to further reduce any LCS into T_1^d. However, we preferred a statement for Theorem 6.1 where only our two main transformations are involved.

Note that, via splitting, the reduction above usually transforms T into *several* topologies. All of them must be T_2^d or LCS's for T to have decidable reachability.

The "$\Leftarrow$" direction is immediate in view of Theorems 4.5.1 and 5.1.

For the "$\Rightarrow$" direction, we can assume w.l.o.g. that T is *reduced*, i.e., it cannot be split as some $T_1 \triangleright T_2$, and it does not contain any reliable essential channel (that could be fused).

We now assume, by way of contradiction, that T cannot be transformed, via general fusions, to T_2^d or to a LCS. From this we show that reachability is not decidable for T. When showing this, we sometimes use three additional transformations ("simplification", "doubling of loops" and "non-essential fusion") that are described in the full version of this paper. We now start an involved case analysis.

1. Since T cannot be transformed to a LCS, it contains a reliable channel c_r, linking node $A = s(c_r)$ to node $B = d(c_r)$. We can assume $A \neq B$, otherwise T contains T_1^u (from Fig. 2) and we conclude immediately with undecidability.

2. T must contain a path p of the form $A = P_0, c_1, P_1, c_2, \ldots, c_n, P_n = B$ that links A to B without using c_r, otherwise c_r would be essential, contradicting the assumption that T is reduced. We pick the shortest such p (it is a simple path) and we call T' the subgraph of T that only contains p, c_r, and the nodes to which they connect.

3. If all c_i's along p are reliable, T' can be transformed to T_2^u (from Fig. 2) by reliable fusions, hence T', and then T itself, have undecidable reachability. Therefore we can assume that at least one c_i along p is lossy.

4. Assume that there exist two nodes P_i, P_j along p that are connected via a third path p' disjoint from c_r and p. We put no restrictions on the relative positions of P_i and P_j but we assume that p' is not a trivial empty path if $i = j$. In that case, let T'' be the subgraph of T that contains c_r, p, and p', and where all channels except c_r are downgraded to lossy if they were reliable. Using simplification and doubling of lossy loops, T'' can be transformed to an undecidable topology among $\{T_3^u, T_4^u, T_5^u, T_6^u\}$. Hence T'' does not have decidable reachability. Neither has T since taking subgraphs and downgrading channels can only improve decidability.

5. If we are not in case 4, the nodes along p do not admit a third path like p'. Therefore all channels along p must be lossy, since we assumed T is reduced. Thus T' can be transformed to T_2^d by general fusion. Since we assumed T cannot be transformed to T_2^d, T must contain extra nodes or channels beyond those of T'. In particular, this must include extra nodes since we just assumed that T has no third path p' between the T' nodes. Furthermore these extra nodes must be connected to the T' part otherwise splitting T would be possible. There are now several cases.

6. We first consider the case of an extra node C with a reliable channel c from C to T'. Since T is reduced, c is not essential and there must be a second path p' from C to T'. Call T'' the subgraph of T that only contains T', C, c and p'. Applying non-essential fusion on c, p' becomes a path between some P_i, P_j and we are back to case 4. Hence undecidability.

7. Next is the case of an extra node C with a reliable channel c from T' to C. Again, since c is not essential, there must be a second path p' from T' to C. Again, the induced subgraph T'' can be shown undecidable as in case 6, reducing to case 4.

8. If there is no extra node linked to T' via a reliable c, the extra nodes must be linked to T' via lossy channels. Now the connection must go both ways, otherwise splitting would be possible. The simplest case is an extra node C with a lossy c from C to T' and a lossy c' from T' to C. But this would have been covered in case 4.

9. Finally there must be at least two extra nodes C and C', with a lossy channel c from C to T' and a lossy c' from T' to C'. We can assume that all paths between T' and C, C' go through c and c', otherwise we would be in one of the cases we already considered. Furthermore C and C' must be connected otherwise T could be split. There are several possibilities here.

10. If there is a path from C' to C we are back to case 4. Hence undecidability.

11. Thus all paths connecting C and C' go from C to C'. If one such path is made of reliable channels only, reliable fusion can be applied on the induced subgraph, merging C and C' and leading to case 8 where undecidability has been shown. If they all contain one lossy channel, T can be split, contradicting our assumption. that it is reduced.

We have now covered all possibilities when T is reduced but cannot be transformed to a LCS or to T_2^d. In all cases is has been shown that reachability is not decidable for T. This concludes the proof of Theorem 6.1.

7 A Classification Algorithm

Theorem 7.1 (Polynomial-time classification). *There exists a polynomial-time algorithm that classifies topologies according to whether they have decidable reachability.*

The algorithm relies on Theorem 6.1:

Stage 1: Starting from a topology T, apply splitting and *reliable* fusion as much as possible. When several transformations are possible, pick any of them nondeterministically. At any step, the transformation reduces the size of the topologies at hand, hence termination is guaranteed in a linear number of steps. At this stage we preserved decidability in both directions, hence T has decidability iff all the reduced topologies $T_1, \ldots, T_n$ have.

Stage 2: Each T_i is now simplified using general fusion (not just reliable fusion). If this ends with a LCS or with T_2^d, decidability for T_i has been proved. When fusion can be applied in several ways, we pick one nondeterministically: a consequence of Theorem 6.1's proof is that these choices lead to the same conclusion when starting from a system that cannot be reduced with splitting or reliable fusion. Thus stage 2 terminates in a linear number of steps. When it terminates, either every T_i has been transformed into a LCS or T_2^d, and we conclude that reachability is decidable for T, or one T_i remains unsimplified and we conclude that reachability is not decidable for T.

We observe that when stage 1 finishes, there will never be any new opportunity for reliable fusion or for splitting since stage 2, i.e., general fusion, does not create or destroy any path between nodes.

8 Concluding Remarks

Summary. We introduced *mixed channel systems*, i.e., fifo channel systems where both lossy and reliable channels can be combined in arbitrary topologies. These systems are a generalization of the lossy channel system model (where all channels are lossy and where reachability is decidable) and of the standard model (with unbounded reliable fifo channels, where reachability is undecidable).

For mixed systems, we provide a complete classification of the network topologies according to whether they lead to decidable reachability problems or not. The main tool are reductions methods that transform a topology into simpler topologies with an equivalent decidability status. These reductions end with simple basic topologies for which the decidability status is established in Section 3.

Directions for future work. At the moment our classification is given implicitly, via a simplification procedure. A more satisfactory classification would be a higher-level description, in the form of a structural criterion, preferably expressible in logical form (or via excluded minors, ...). Obtaining such a description is our more pressing objective.

Beyond this issue, the two main avenues for future work are extending the MCS model (e.g., by considering other kinds of unreliability in the style of [9], or by allowing guards in the style of [5], etc.) and considering questions beyond just reachability and safety (e.g., termination and liveness).

References

1. Abdulla, P.A., Baier, C., Purushothaman Iyer, S., Jonsson, B.: Simulating perfect channels with probabilistic lossy channels. Information and Computation 197(1–2), 22–40 (2005)
2. Abdulla, P.A., Bertrand, N., Rabinovich, A., Schnoebelen, Ph.: Verification of probabilistic systems with faulty communication. Information and Computation 202(2), 141–165 (2005)
3. Abdulla, P.A., Collomb-Annichini, A., Bouajjani, A., Jonsson, B.: Using forward reachability analysis for verification of lossy channel systems. Formal Methods in System Design 25(1), 39–65 (2004)
4. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Information and Computation 127(2), 91–101 (1996)
5. Baier, C., Bertrand, N., Schnoebelen, Ph.: On computing fixpoints in well-structured regular model checking, with applications to lossy channel systems. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 347–361. Springer, Heidelberg (2006)
6. Baier, C., Bertrand, N., Schnoebelen, Ph.: Verifying nondeterministic probabilistic channel systems against ω-regular linear-time propertie. ACM Trans. Computational Logic 9(1) (2007)

7. Bouyer, P., Markey, N., Ouaknine, J., Schnoebelen, Ph., Worrell, J.: On termination for faulty channel machines. In: Proc. STACS 2008, pp. 121–132 (2008)
8. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM 30(2), 323–342 (1983)
9. Cécé, G., Finkel, A., Purushothaman Iyer, S.: Unreliable channels are easier to verify than perfect channels. Information and Computation 124(1), 20–31 (1996)
10. Chambart, P., Schnoebelen, Ph.: Post embedding problem is not primitive recursive, with applications to channel systems. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 265–276. Springer, Heidelberg (2007)
11. Chambart, P., Schnoebelen, Ph.: The ω-regular Post embedding problem. In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 97–111. Springer, Heidelberg (2008)
12. Chambart, P., Schnoebelen, Ph.: The ordinal recursive complexity of lossy channel systems. In: Proc. LICS 2008, IEEE Comp.Soc. Press, Los Alamitos (2008)
13. Finkel, A.: Decidability of the termination problem for completely specificied protocols. Distributed Computing 7(3), 129–135 (1994)
14. Henriksen, J.G., Mukund, M., Kumar, K.N., Sohoni, M.A., Thiagarajan, P.S.: A theory of regular MSC languages. Information and Computation 202(1), 1–38 (2005)
15. Jurdziński, M., Lazić, R.: Alternation-free modal mu-calculus for data trees. In: Proc. LICS 2007, pp. 131–140. IEEE Comp. Soc. Press, Los Alamitos (2007)
16. Kurucz, A.: Combining modal logics. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) Handbook of Modal Logics, ch. 15, vol. 3, pp. 869–926. Elsevier Science, Amsterdam (2006)
17. Lasota, S., Walukiewicz, I.: Alternating timed automata. ACM Trans. Computational Logic 9(2) (2008)
18. Lohrey, M., Muscholl, A.: Bounded MSC communication. Information and Computation 189(2), 160–181 (2004)
19. Ouaknine, J., Worrell, J.: On the decidability and complexity of Metric Temporal Logic over finite words. Logical Methods in Comp. Science 3(1), 1–27 (2007)
20. Pachl, J.K.: Protocol description and analysis based on a state transition model with channel expressions. In: Proc. PSTV 1987, pp. 207–219. North-Holland, Amsterdam (1987)
21. Schnoebelen, Ph.: The verification of probabilistic lossy channel systems. In: Baier, C., et al. (eds.) Validation of Stochastic Systems – A Guide to Current Research. LNCS, vol. 2925, pp. 445–465. Springer, Heidelberg (2004)

On the Reachability Analysis of Acyclic Networks of Pushdown Systems*

Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili

LIAFA, CNRS & Univ. of Paris 7, Case 7014, 75205 Paris 13, France
{atig,abou,touili}@liafa.jussieu.fr

Abstract. We address the reachability problem in acyclic networks of pushdown systems. We consider communication based either on shared memory or on message passing through unbounded lossy channels. We prove mainly that the reachability problem between recognizable sets of configurations (i.e., definable by a finite union of products of finite-state automata) is decidable for such networks, and that for lossy channel pushdown networks, the channel language is effectively recognizable. This fact holds although the set of reachable configurations (including stack contents) for a network of depth (at least) 2 is not rational in general (i.e., not definable by a multi-tape finite automaton). Moreover, we prove that for a network of depth 1, the reachability set is rational and effectively constructible (under an additional condition on the topology for lossy channel networks).

1 Introduction

The verification of concurrent programs is an important and highly challenging problem. It is well known that basic analysis problems (e.g., control point reachability) on concurrent programs with (recursive) procedure calls are undecidable in general (even for programs with boolean data). A lot of efforts have been nevertheless devoted recently to the development of (1) precise analysis algorithms for some particular program models [8, 9, 10, 12, 16, 18, 21, 22, 27], and of (2) generic analysis techniques based on computing approximations of the set of program behaviors [6, 7, 13, 24, 28].

In this paper we address the issue of analyzing concurrent programs with *asynchronous acyclic communication*, i.e., programs where the communication relation between parallel processes has a *directed acyclic graph* (i.e., the information flows only in one direction between any two processes, but it can follow several paths). We consider two kinds of communication mechanisms: communication using a shared memory, and communication through unreliable fifo channels. We define models of such concurrent programs, and we investigate the decidability of their reachability problem, as well as the issue of constructing a finite representation of their sets of reachable configurations.

The program models we define are based on *networks of communicating pushdown systems*. It is indeed well admitted that pushdown systems are adequate models for sequential programs with recursive procedure calls [15, 26], and therefore, its is natural to model concurrent programs as parallel communicating pushdown processes.

We start by considering asynchronous communication with shared memory. We define a model, called APN_{obs}, which consists in a finite collection of pushdown processes

* Supported by the French projects ANR-06-SETI-001 AVERISS and RNTL AVERILES.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 356–371, 2008.

communicating according to an *acyclic observation relation R*: a process P can observe (i.e., read and test) the control state of a process Q if $(P,Q) \in R$. Intuitively, the control state of a process Q represents the content of the variables owned (accessible in read/write modes) by Q, and $(P,Q) \in R$ means that P can read the variables owned by Q. (Considering cyclic observation relations leads to Turing powerful models).

The configurations of an $\mathrm{APN}_{\mathsf{obs}}$ with n processes $P_1,\ldots,P_n$ are n-dim vectors of words of the form $p_i w_i$, where p_i (resp. w_i) represents the control state (resp. the stack content) of the process P_i. Therefore, sets of configurations are n-dim languages (i.e., sets of n-dim vectors of words), and we need to consider finite representations for such languages. Two natural and well-known classes of n-dim languages are (1) the class of *rational* languages definable by multi-tape finite automata, and (2) the class of *recognizable* languages which are finite unions of products of regular (1-dim) word languages. (Every recognizable language is a rational one, but the converse is not true in general. For instance, the set $\{(a^n,b^n) : n \geq 0\}$ is rational but not recognizable).

Then, we are interested in the problem of deciding whether, given two sets of configurations S_1 and S_2, supposed to be recognizable or rational (effectively represented, respectively, either by a multi-tape automaton, or by a finite collection of vectors of finite automata), it is possible to reach a configuration in S_2 from a configuration in S_1. We are also interested in the nature of the sets of reachable configurations in the sense that we want to determine whether these sets are recognizable or rational and effectively constructible, or they are outside these classes.

We show that the set of reachable configurations in an $\mathrm{APN}_{\mathsf{obs}}$ from a single configuration is *not* rational in general, and this fact holds as soon as the graph of the observation relation is of depth 2. We also prove that the reachability problem from a single configuration to a rational set of configurations is *undecidable* for such networks. On the other hand, we establish several positive results when the source and target sets in the reachability problem are recognizable. First, we prove that for networks of depth 1, the set of reachable configurations from a recognizable set (which is not recognizable in general) is always *rational* and constructible. (From this result follows the decidability of the reachability problem between recognizable sets for networks of depth 1.) Furthermore, we show that for the general case (i.e., networks of *any* depth), the reachability problem between recognizable sets of configurations is *decidable*, although the reachability sets for these models are not always rational as mentioned above.

Then, we pursue our study by considering message-passing communication. We introduce the model $\mathrm{APN}_{\mathsf{lc}}$ which is an *acyclic* network of pushdown processes communicating through *unbounded lossy FIFO channels*. A configuration of an $\mathrm{APN}_{\mathsf{lc}}$ consists in a vector of local configurations of each of the processes in the network (i.e., the vector of their control states and stack contents), and a vector of words corresponding to the contents of the channels. $\mathrm{APN}_{\mathsf{lc}}$'s are actually more general than $\mathrm{APN}_{\mathsf{obs}}$'s: every $\mathrm{APN}_{\mathsf{obs}}$ can be simulated by an $\mathrm{APN}_{\mathsf{lc}}$, whereas the converse holds (only) for a linear topology. (Cyclic lossy channel pushdown networks are Turing powerful).

We address for the $\mathrm{APN}_{\mathsf{lc}}$'s the same questions (reachability problem, characterization of the reachability sets) as for the $\mathrm{APN}_{\mathsf{obs}}$'s. Clearly, negative results stating undecidability or non recognizability/rationality for $\mathrm{APN}_{\mathsf{obs}}$'s can be transfered to $\mathrm{APN}_{\mathsf{lc}}$'s. Moreover, we show that, contrary to the case of $\mathrm{APN}_{\mathsf{lc}}$'s, even for networks of depth 1

the set of reachable configurations is *not rational* in general. However, we prove that for networks of depth 1, the set of reachable configurations is rational and constructible when the undirected graph of the communication relation is a forest. Moreover, we prove that the reachability problem between recognizable sets is actually *decidable* for the *whole* class of APN_{lc}'s. We also prove that the *channel language* of APN_{lc}'s, i.e., the *projection* of the set of their reachable configurations on control states and on channel contents, is an *effectively constructible* recognizable set.

For lack of space, detailed proofs are omitted here. They can be found in [4].

Related work: Several models for concurrent programs have been proposed recently, and their verification problems have been investigated. Decidability results in this context appear for instance in [8, 9, 10, 11, 19, 22, 27, 28]. These results do not cover the class of models we consider in this paper.

A form of acyclic observation between pushdown systems was introduced in [8]: a process can observe the states of the processes it has created (dynamic creation is allowed in [8]); however, the process cannot distinguish between different states in a control loop of any observed process. This restriction guarantees that the set of (backward) reachable configurations (of the models defined in [8]) is definable using a finite tree automaton. In the context of our present work, we can show that a similar restriction guarantees the recognizability of the set of reachable configurations.

In [28], the decidability of the reachability problem is established for a finite number of computation phases in multi-stack systems, where in each phase the system can pop from one distinguished stack, and push on some number of stacks. Thus, for each pair of stacks s_1 and s_2, the alternation between phases where s_1 is popped while s_2 is pushed, and phases where the converse holds, is bounded. In our models, it is possible to have an unbounded number of such alternations. On the other hand, since the communication relation in our models is fixed, our models cannot simulate phase switches in the sense of [28]. Actually, we can prove that in our APN_{obs} models, switching between different communication relations leads to an undecidable model, even for one single switch. (The proof is by a simple reduction of PCP [4]).

Many works have addressed the verification problem of lossy FIFO channel systems with a finite control structure (see, e.g., [2,3]). For this case, the reachability problem is decidable (for any network topology) and the proof uses the theory of monotonic systems w.r.t. a well-quasi ordering on the configuration space [1,17]. Arguments based on this theory are not applicable to our models (due to the stacks). Moreover, the channel language of finite-control lossy channel systems is recognizable but non constructible in general [23]. We prove here that the channel language is constructible for acyclic network with a pushdown-definable control. Also, the complexity of the reachability problem for finite-control lossy channel systems is nonprimitive recursive [25]. In our case, we have established a primitive recursive upper-bound.

Acyclic networks of FIFO channel systems have been considered in [14] where the authors prove that for two finite-control processes communicating with one perfect channel and one lossy channel the reachability problem is decidable.

In [12], the authors consider unidirectional lossy channel pushdown systems. They prove the decidability of the reachability problem under the restriction that each process can read a message from a channel *only* when its stack is empty.

Networks of pushdown systems communicating through *perfect* FIFO-channel systems have been considered in [20]. The reachability problem is decidable for a finite number of phases where in each phase only one process is allowed to run, and this process can read from one single *input channel*, but *only if its stack is empty* (like in [12]), and to send to output channels *different from the input channel*. Actually, the authors show that these models can be simulated by the ones they have considered in [28]. Again, these models are not comparable with our $\mathrm{APN}_{\mathrm{lc}}$ models. In particular, we do not require that a message reception can occur only if the stack is empty. Moreover, in the framework of [20], since the channels are perfect, allowing a process to receive messages from two different channels leads to a Turing powerful model, whereas reception from several channels is allowed in our case.

2 Preliminaries

2.1 Languages and Finite Automata

Let Σ be a finite alphabet. We denote by Σ^* (resp. Σ^+) the set of all *words* (resp. non empty words) over Σ, and by ε the empty word. A language is a (possibly infinite) set of words. Let $w = a_1 \dots a_n$ be a word in Σ^*, then the reverse of w is the word $w^R = a_n \dots a_1$. Let $\Sigma_1, \dots, \Sigma_n$ be n finite alphabets. A n-dim word over $\Sigma_1, \dots, \Sigma_n$ is an element of $\Sigma_1^* \times \dots \times \Sigma_n^*$. A n-dim language is a (possibly infinite) set of n-dim words.

Given two n-dim words $\mathbf{u} = (u_1, \dots, u_n)$ and $\mathbf{v} = (v_1, \dots, v_n)$, their concatenation is defined by $\mathbf{uv} = (u_1 v_1, \dots, u_n v_n)$. Let $\Sigma_\varepsilon = \Sigma_1 \cup \{\varepsilon\} \times \dots \times \Sigma_n \cup \{\varepsilon\}$. It easy to see that every n-dim word is a concatenation of elements of Σ_ε.

Given an alphabet Σ, we denote by $\preceq \subseteq \Sigma^* \times \Sigma^*$ the *subword relation* defined as follows: for every $u = a_1 \cdots a_n \in \Sigma^*$, and every $v = b_1 \cdots b_m \in \Sigma^*$, $u \preceq v$ iff $\exists i_1, \dots, i_n \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_n$ and $\forall j \in \{1, \dots, n\}, a_j = b_{i_j}$. The relation $\preceq$ is generalized in a pointwise manner to n-dim words as follows: Let $\mathbf{u} = (u_1, \dots, u_n)$ and $\mathbf{v} = (v_1, \dots, v_n)$ be two n-dim words, $\mathbf{u} \preceq \mathbf{v}$ iff for every i, $1 \leq i \leq n$, $u_i \preceq v_i$. $\prec$ is the strict subword relation: $\mathbf{u} \prec \mathbf{v}$ iff $\mathbf{u} \preceq \mathbf{v}$ and $\mathbf{u} \neq \mathbf{v}$.

Given a (n-dim) language $L \subseteq \Sigma_1^* \times \dots \times \Sigma_n^*$, the *upward closure* (resp. *downward closure*) of L (w.r.t. $\preceq$) is the set $L{\uparrow}$ (resp. $L{\downarrow}$) $= \{\mathbf{u} \in \Sigma_1^* \times \dots \times \Sigma_n^* : \exists \mathbf{v} \in L.\ \mathbf{v} \preceq \mathbf{u}$ (resp. $\mathbf{u} \preceq \mathbf{v}$) $\}$. A (n-dim) language L is upward closed (resp. downward closed) iff $L{\uparrow} = L$ (resp. $L{\downarrow} = L$).

A *n-tape automaton* over $\Sigma_1, \dots, \Sigma_n$ is a tuple $T = (Q, \Sigma_1, \dots, \Sigma_n, \delta, I, F)$ where Q is a finite set of states, $\delta \subseteq Q \times \Sigma_\varepsilon \times Q$ is a labeled transition relation, $I \in Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. Given a *n-tape* automaton $T = (Q, \Sigma_1, \dots, \Sigma_n, \delta, I, F)$ and a state $q \in Q$, let T^q denote the *n-tape* automaton $(Q, \Sigma_1, \dots, \Sigma_n, \delta, \{q\}, F)$. A run of T over $\mathbf{w} \in \Sigma_1^* \times \dots \times \Sigma_n^*$ is a states sequence $q_0 q_1 \cdots q_m \in Q^+$ such that (1) $q_0 \in I$, (2) $\exists \mathbf{u_0}, \dots, \mathbf{u_{m-1}} \in \Sigma_\varepsilon$. $\forall i \in \{0, \dots, m-1\}$. $(q_i, \mathbf{u_i}, q_{i+1}) \in \delta$ and $\mathbf{u_0} \cdots \mathbf{u_{m-1}} = \mathbf{w}$. The run is *accepting* if $q_m \in F$. The language of T, denoted $L(T)$, is the set of n-dim words for which there is an accepting run of T.

A n-dim language is *rational* if it is definable as the language of some n-tape automaton. Note that 1-tape automata are the usual finite-state word automata. Their languages are commonly known to be *regular*. A n-dim language L is *recognizable* if it is a finite union of products of n regular languages (i.e. $L = \bigcup_{j=1}^{m} L(A_1^j) \times \dots \times L(A_n^j)$ for some

$m \in \mathbb{N}$, where A_i^j is an automaton over Σ_i). The class of rational languages subsumes strictly the class of recognizable languages. For instance, the set $\{(a^n, b^n) : n \geq 0\}$ is rational but not recognizable, whereas $\{(a^n, b^m) : n, m \geq 0\}$ is recognizable, and $\{(a^n, b^n c^n) : n \geq 0\}$ is not rational.

Let us recall some well known facts about these classes of languages (see, e.g., [5]), and fix some notations. First, the class of recognizable languages, for any dimension $n \geq 1$, is closed under boolean operations. On the other hand, for every $n \geq 2$, the class of n-dim rational languages is closed under union, but not under complementation, nor under intersection. However, the intersection of a rational language with a recognizable language is rational. The emptiness problem of n-tape automata is decidable, and the same holds for the inclusion problem of recognizable languages. However, the inclusion problem is undecidable for rational languages (for $n \geq 2$).

Rational languages are also closed under projection, defined as follows: Given a n-tape automaton T over $\Sigma_1, \ldots, \Sigma_n$, and a set of indices $\iota \subset \{1, \ldots, n\}$, the projection of T on ι, denoted $\Pi_\iota(T)$, is the automaton obtained by erasing all the tapes which are not in ι (if ι has k indices, then $\Pi_\iota(T)$ is a k-tape automaton). Rational languages are also closed under composition: Let T and T' be two multi-tape automata on, respectively, the alphabets $\Sigma_1, \ldots, \Sigma_n$ and $\Sigma'_1, \ldots, \Sigma'_m$, and let $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$ be two indices s.t. $\Sigma'_j = \Sigma_i$. Then, it is possible to construct a $(n+m-1)$-tape automaton $T \circ_{(i,j)} T'$ which accepts $(w_1, \ldots, w_n, w'_1, \ldots, w'_{j-1}, w'_{j+1}, \ldots, w'_m)$ iff $(w_1, \ldots, w_n) \in L(T)$ and $(w'_1, \ldots, w_{j-1}, w_i, w_{j+1}, \ldots, w'_m) \in L(T')$, i.e. the composition corresponding to the synchronization of the i^{th} tape of T with the j^{th} tape of T'.

We also define a composition operator $\diamond_{(i,j)}$ such that, given two multi-tape automata T and T', if the content of the j^{th} tape of T' (say w'_j) is a prefix of the one of the i^{th} tape of T (say w_i), then the i^{th} tape of $T \diamond_{(i,j)} T'$ contains the word w s.t. $w_i = w'_j w$. Formally, assume that T and T' are defined on the alphabets $\Sigma_1, \ldots, \Sigma_n$ and $\Sigma'_1, \ldots, \Sigma'_m$ respectively, and let $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$ such that $\Sigma'_j = \Sigma_i$. Then, it is possible to construct a $(n+m-1)$-tape automaton $T \diamond_{(i,j)} T'$ that accepts the set of vectors $(w_1, \ldots, w_{i-1}, w, w_{i+1}, \ldots, w_n, w'_1, \ldots, w'_{j-1}, w'_{j+1}, \ldots, w'_m)$ if $\exists w_i, w'_j \in \Sigma_i^*$ such that: (1) $(w_1, \ldots, w_n) \in L(T)$, (2) $(w'_1, \ldots, w'_m) \in L(T')$, and (3) $w_i = w'_j w$.

Proposition 1. *Let $T = (Q, \Sigma_1, \ldots, \Sigma_n, \delta, I, F)$ be a n-tape automaton. Then, $L(T)\downarrow$ and $L(T)\uparrow$ are recognizable and effectively definable by the union of $((|\Sigma_1|+1) \times \cdots \times (|\Sigma_n|+1))^{|Q|}$ products of n finite state automata with at most $|Q|$ states.*

2.2 Labeled Pushdown Systems

A Labeled Pushdown System (LPDS) is defined by a tuple $\mathcal{P} = (P, \Sigma, \Gamma, \Delta)$ where P is a finite set of states, Σ is the input alphabet (actions), Γ is the stack alphabet, and Δ is a finite set of transition rules of the form $\langle p, u \rangle \overset{a}{\hookrightarrow} \langle p', u' \rangle$; where: (1) $p, p' \in P$, (2) $a \in \Sigma \cup \{\varepsilon\}$, and (3) $u, u' \in \Gamma^*$ s.t. either (*i*) $|u| = 1$ and $u' = \varepsilon$ (pop operation), (*ii*) $u = \varepsilon$ and $|u'| = 1$ (push operation), or (*iii*) $u = u' = \varepsilon$ (no operation on the stack).

A *configuration* of an LPDS is a word $pw \in P\Gamma^*$ where p is a state and w is a stack content. In particular, configurations of the form $p\varepsilon$ are simply denoted by p. We define a *transition relation* $\overset{a}{\Longrightarrow}_{\mathcal{P}}$ between configurations as follows: $pw \overset{a}{\Longrightarrow}_{\mathcal{P}} p'w'$

if $\exists\ (\langle p,u\rangle \overset{a}{\hookrightarrow} \langle p',u'\rangle) \in \Delta$ and $\exists v \in \Gamma^*$ s.t. $w = uv$ and $w' = u'v$. We generalize the transition relation $\overset{a}{\Longrightarrow}_{\mathcal{P}}$ to sequences of actions in the usual way: $pw \overset{\varepsilon}{\Longrightarrow}_{\mathcal{P}} pw$ for every $pw \in P\Gamma^*$, and given a word $\sigma = a_0 \cdots a_{n-1} \in \Sigma^+$, $pw \overset{\sigma}{\Longrightarrow}_{\mathcal{P}} p'w'$ iff $\exists p_0w_0, \ldots, p_nw_n \in P\Gamma^*$ s.t. $pw = p_0w_0$, $p'w' = p_nw_n$, and $p_iw_i \overset{a_i}{\Longrightarrow}_{\mathcal{P}} p_{i+1}w_{i+1}$ for every $i \in \{0,\ldots,n-1\}$.

A set of configurations $C \subseteq P\Gamma^*$ is recognizable iff it is recognized by some finite state automaton (i.e., there exists an automaton $\mathcal{A}$ such that $C = L(\mathcal{A})$).

Given an LPDS $\mathcal{P}$ and two recognizable sets of configurations $C, C' \subseteq P\Gamma^*$, let $Traces_{\mathcal{P}}(C,C') = \{\sigma \in \Sigma^* : \exists (c,c') \in C \times C', c \overset{\sigma}{\Longrightarrow}_{\mathcal{P}} c'\}$ be the set of sequences that lead $\mathcal{P}$ from C to C'. Clearly, $Traces_{\mathcal{P}}(C,C')$ is a context-free language, and conversely, every context-free language can be defined as a trace language of some LPDS.

Proposition 2. *Let $\mathcal{P} = (P,\Sigma,\Gamma,\Delta)$ be an LPDS and $p, p' \in P\Gamma^*$ be two configurations. It is possible to construct, in exponential (resp. double exponential) time in $(|P| + |\Sigma| + |\Gamma|)$, a finite state automaton $A = (Q,\Sigma,\delta,I,F)$ such that $L(A) = Traces_{\mathcal{P}}(p,p')\downarrow$ (resp. $L(A) = Traces_{\mathcal{P}}(p,p')\uparrow$), where in the worst case, $|Q|$ is exponential (resp. doubly exponential) in $(|P| + |\Sigma| + |\Gamma|)$.*

3 Relating Reachable Configurations with Traces in LPDS

We establish hereafter a result that is a key ingredient for the construction of Section 5. More precisely, let $\mathcal{P} = (P,\Sigma,\Gamma,\Delta)$ be a LPDS, and let $p \in P$. We show hereafter that the sets $U(\mathcal{P},p)$ and $D(\mathcal{P},p)$ defined below are rational and effectively constructible:

$$U(\mathcal{P},p) = \{(p'w,\sigma) \in P\Gamma^* \times \Sigma^* : \sigma^R \in Traces_{\mathcal{P}}(p,p'w)\uparrow\} \quad (1)$$

$$D(\mathcal{P},p) = \{(p'w,\sigma) \in P\Gamma^* \times \Sigma^* : \sigma^R \in Traces_{\mathcal{P}}(p,p'w)\downarrow\} \quad (2)$$

Theorem 1. *It is possible to construct a 2-tape automaton $\widetilde{T}(\mathcal{P},p)$ (resp. $\widehat{T}(\mathcal{P},p)$), with a number of states that is at most exponential (resp. doubly exponential) in $(|P| + |\Sigma| + |\Gamma|)$, such that $L(\widetilde{T}(\mathcal{P},p)) = D(\mathcal{P},p)$ (resp. $\widehat{T}(\mathcal{P},p) = U(\mathcal{P},p)$).*

We sketch hereafter the construction of a 2-tape automaton $\widetilde{T}(\mathcal{P},p)$ that accepts the set $D(\mathcal{P},p)$ (the one for $U(\mathcal{P},p)$ is similar). We define of a 2-tape automaton $T = (Q,\Gamma,\Sigma,\delta,I,F)$ with $I = P$, such that (w,σ) is accepted by T from the state p' iff $(p'w,\sigma) \in L(\widetilde{T}(\mathcal{P},p))$, i.e., $L(\widetilde{T}(\mathcal{P},p)) = \bigcup_{p' \in P}(p',\varepsilon)L(T^{p'})$. The construction of T is as follows: Consider a computation $p \overset{\sigma'}{\Longrightarrow}_{\mathcal{P}} p'\gamma_m \cdots \gamma_1$ of $\mathcal{P}$ with $p,p' \in P$, $\sigma' \in \Sigma^*$, and $\gamma_1,\ldots,\gamma_m \in \Gamma$. This computation can be decomposed as follows: $\exists p_0,p'_0,p_1,p'_1,\ldots,p_m \in P$, $\exists \sigma'_0,\ldots,\sigma'_m \in \Sigma^*$, and $\exists a'_0,\ldots,a'_{m-1} \in (\Sigma \cup \{\varepsilon\})$ such that: (1) $p_0 = p$, (2) $\sigma' = \sigma'_0a'_0\sigma'_1a'_1 \cdots \sigma'_{m-1}a'_{m-1}\sigma'_m$, (3) $\forall i \in \{0,\ldots,m-1\}$, $p_i \overset{\sigma'_i}{\Longrightarrow}_{\mathcal{P}} p'_i$ (i.e. $\sigma'_i \in Traces_{\mathcal{P}}(p_i,p'_i)$) and $(\langle p'_i,\varepsilon\rangle \overset{a'_i}{\hookrightarrow} \langle p_{i+1},\gamma_{i+1}\rangle) \in \Delta$, and (4) $p_m \overset{\sigma'_m}{\Longrightarrow}_{\mathcal{P}} p'$.

Now, let σ be a word over Σ such that $\sigma \preceq \sigma'$. Then, $\exists \sigma_0,\ldots,\sigma_m \in \Sigma^*$ and $\exists a_0,\ldots,a_{m-1} \in (\Sigma \cup \{\varepsilon\})$ such that: (1) $\forall i \in \{0,\ldots,m\}$, $\sigma_i \preceq \sigma'_i$ (i.e. $\sigma_i \in Traces_{\mathcal{P}}(p_i,p'_i)\downarrow$), and (2) $\forall i \in \{0,\ldots,m-1\}$, $a_i \preceq a'_i$ (i.e., a_i is either a'_i or ε). Then, we have: $(\gamma_m \cdots \gamma_1, \sigma^R) = (\varepsilon,\sigma_m^R)(\gamma_m,a_{m-1})(\varepsilon,\sigma_{m-1}^R)(\gamma_{m-1},a_{m-2}) \cdots (\gamma_1,a_0)(\varepsilon,\sigma_0^R)$.

The 2-tape automaton T must recognize such pairs $(\gamma_m \cdots \gamma_1, \sigma^R)$. Therefore to construct T we proceed as follows: (1) For each pair $(q,q') \in P \times P$, we construct the automaton $\mathcal{A}_{(q,q')}$ that recognizes the set $Traces_{\mathcal{P}}(q,q')\downarrow$ such that q (resp. q') is its unique initial (resp. final) state. This automaton is effectively constructible due to Proposition 2. (2) We consider automata recognizing the mirror (reverse) language of the automata $\mathcal{A}_{(q,q')}$ and extend them to 2-tape words by adding ε on the first tape on all transitions. (3) We connect these automata using transitions of the forms $(p_{i+1},(\gamma_{i+1},a'_i),p'_i)$ and $(p_{i+1},(\gamma_{i+1},\varepsilon),p'_i)$, for every rule $\langle p'_i,\varepsilon\rangle \overset{a'_i}{\hookrightarrow} \langle p_{i+1},\gamma_{i+1}\rangle$ in Δ (note that p_{i+1} is the initial state of $\mathcal{A}_{(p_{i+1},p'_{i+1})}$, and p'_i the final state of $\mathcal{A}_{(p_i,p'_i)}$).

4 Acyclic Networks of Pushdown Systems

An Acyclic Observation Relation Pushdown Network (APN_{obs} for short) is defined by a tuple $N = (\mathcal{P}_1,\ldots,\mathcal{P}_n,R)$ where $R \subseteq \{(i,j) : 1 \leq i < j \leq n\}$ is an *antisymmetric* binary relation (R defines an acyclic directed graph whose nodes are $1,\ldots,n$), and $\forall i \in \{1,\ldots,n\}$, $\mathcal{P}_i = (P_i,\Gamma_i,\Delta_i)$ is a (constrained) pushdown system where P_i is a finite set of control states, Γ_i is a finite stack alphabet, and Δ_i is a finite set of transition rules of the form $\phi : \langle p,u\rangle \hookrightarrow \langle p',u'\rangle$ where (1) $\phi \subseteq \bigcup_{(i,j)\in R} P_j$, (2) $p,p' \in P_i$, and (3) $u,u' \in \Gamma_i^*$ such that either (*i*) $|u| = 1$ and $u' = \varepsilon$, (*ii*) $u = \varepsilon$ and $|u'| = 1$, or (*iii*) $u = u' = \varepsilon$.

An APN_{obs} is *linear* if the relation R is of the form $R = \{(i,i+1) : 1 \leq i < n\}$. An APN_{obs} N consisting of a single process $\mathcal{P}$, i.e. $N = (\mathcal{P},\emptyset)$, will be denoted by $\mathcal{P}$.

The *depth* of $\mathcal{P}_i$ in N, denoted $d(i)$, is the length of the longest path starting from i in the graph of the relation R. The depth of N is the maximal depth of its processes.

Our models are networks of pushdown systems where each process can observe other processes according to the relation R: A process $\mathcal{P}_i$ can observe any process $\mathcal{P}_j$ s.t. $(i,j) \in R$. In this case, the execution of a rule of $\mathcal{P}_i$ can be conditioned by the fact that $\mathcal{P}_j$ is at some particular state (specified in the constraint ϕ of the rule).

A *local configuration* of a process in the network, say $\mathcal{P}_i$, is a word $p_i w_i \in P_i\Gamma_i^*$ where p_i is a state and w_i is a stack content. A *configuration* of the network N is a vector $(p_1w_1,\ldots,p_nw_n) \in \prod_{i=1}^n P_i\Gamma_i^*$, where p_iw_i is the local configuration of $\mathcal{P}_i$. (Notice that a vector $(p_1,\ldots,p_n) \in \prod_{i=1}^n P_i$ is a configuration where all processes have empty stacks).

We define a *transition relation* $\Longrightarrow_N$ between configurations as follows: $(p_1w_1,\ldots,p_nw_n) \Longrightarrow_N (p'_1w'_1,\ldots,p'_nw'_n)$ if $\exists i \in \{1,\ldots,n\}$, and $\exists(\phi : \langle p,u\rangle \hookrightarrow \langle p',u'\rangle) \in \Delta_i$ such that (1) $p = p_i$ and $p' = p'_i$, (2) $w_i = uv$ and $w'_i = u'v$ for some $v \in \Gamma_i^*$, (3) $\forall j > i$, if $(i,j) \in R$, then $p_j \in \phi$, and (4) $\forall j \neq i$. $p_j = p'_j$ and $w_j = w'_j$. Given a configuration $c \in \prod_{i=1}^n P_i\Gamma_i^*$, the set of immediate successors of c is $post_N(c) = \{c' \in \prod_{i=1}^n P_i\Gamma_i^* : c \Longrightarrow_N c'\}$. This definition is generalized to sets of configurations in the usual manner. $post_N^*$ denotes the reflexive-transitive closure of $post_N$.

Proposition 3. *The reachability problem for APN_{obs}'s can be reduced to the reachability problem for linear APN_{obs}'s.*

Remark 1. The extension of APN_{obs} by allowing cycles in communication relation leads to Turing powerful models.

Remark 2. APN_{obs} of depth zero are collections of independent pushdown systems. Their analysis boils down to the analysis of each of the processes separately. Therefore, we consider in the sequel only networks of depth ≥ 1.

5 Computing $post^*$ Images for APN_{obs}

5.1 Limits of Recognizability and Rationality

Consider a network of depth 2 with three processes $\mathcal{P}_1$ observing $\mathcal{P}_2$ which observes $\mathcal{P}_3$. Assume that $\mathcal{P}_3$ cycles on two different states and pushes in its stack an a at each cycle, and then moves to a termination state. During such a computation, $\mathcal{P}_2$ mimics $\mathcal{P}_3$ by pushing in its stack a b after each (observable) cycle of $\mathcal{P}_3$, and in the same time, $\mathcal{P}_1$ mimics $\mathcal{P}_2$ by pushing a c after each (observable) cycle of $\mathcal{P}_2$. At the end of this phase, $\mathcal{P}_2$ moves to another state where it starts cycling (on two different states) and popping at each cycle a symbol from its stack. During this new phase, $\mathcal{P}_1$ pushes a d at each observable cycle of $\mathcal{P}_2$. Then, the set of reachable configurations when $\mathcal{P}_2$ has emptied its stack is $\{(d^\ell c^m, \varepsilon, a^n) : \ell \leq n, m \leq n\}$, which is not a rational set. A similar example, with two cyclic pushing processes where one of them imitates the other one, shows that the reachability set for networks depth 1 is not recognizable in general.

Proposition 4. *Given an APN_{obs} N of depth ≥ 1 (resp. ≥ 2), and a configuration c of N, the set $post^*_N(c)$ is not recognizable (resp. not rational) in general.*

5.2 Reachability Analysis for APN_{obs} of Depth 1

We prove that for every APN_{obs} of depth 1, the $post^*$ image of any recognizable set of configurations C is a rational set. For proving that we can assume w.l.o.g. that C is reduced to a configuration of the form $(p_1, \ldots, p_n)$ where all the stacks are empty.

The case of two processes: We first present the proof for the special case of a network with two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{P}_1$ observes $\mathcal{P}_2$, i.e., $N = (\mathcal{P}_1, \mathcal{P}_2, \{(1,2)\})$, where $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i)$ and $C = \{(p_1, p_2)\}$. We consider later the general case.

To show that $post^*_N(C)$ is effectively rational, we proceed as follows: First, we compute a 2-tape automaton T_1 that accepts the set U_1 consisting of pairs $(pw, \sigma^R) \in P_1\Gamma_1^* \times P_2^*$ s.t. $\mathcal{P}_1$ can reach the configuration pw from p_1 *provided that* the observed sequence of states of $\mathcal{P}_2$ is σ. Second, we construct a 2-tape automaton T_2 that accepts the set U_2 of pairs $(p'w', \sigma^R) \in P_2\Gamma_2^* \times P_2^*$ such that σ is a sequence of $\mathcal{P}_2$ states that can be observed by $\mathcal{P}_1$ during a computation of $\mathcal{P}_2$ from p_2 to $p'w'$. Then, we have $(pw, p'w') \in post^*_N(C)$ iff $\exists \sigma \in P_2^*$ such that $(pw, \sigma^R) \in U_1$ and $(p'w', \sigma^R) \in U_2$. Hence, a 2-tape automaton T such that $L(T) = post^*_N(C)$ can be obtained by a synchronization operation between T_1 and T_2 on their second tape.

Let $\widehat{\mathcal{P}}_1 = (P_1, P_2, \Gamma_1, \Delta_1')$ be the LPDS such that $\langle p, u\rangle \overset{p_2}{\hookrightarrow} \langle p', u'\rangle \in \Delta_1'$ iff $\exists(\phi : \langle p, u\rangle \hookrightarrow \langle p', u'\rangle) \in \Delta_1$ such that $p_2 \in \phi$. We prove that $U_1 = U(\widehat{\mathcal{P}}_1, p_1)$: Clearly, if there is a run $p_1 \overset{\sigma}{\Longrightarrow}_{\widehat{\mathcal{P}}_1} pw$, then $(pw, \sigma^R) \in U_1$. Moreover, U_1 is *upward closed* in the sense that if $\sigma \in P_2^*$ enables a sequence of transition rules of $\mathcal{P}_1$ that reaches pw from p_1 (i.e.

$(pw, \sigma^R) \in U_1$), then for every $\sigma' \in P_2^*$ such that $\sigma \preceq \sigma'$, $\mathcal{P}_1$ can fire the same sequence of transition rules to reach pw from p_1 (i.e. $(pw, \sigma'^R) \in U_1$). Hence, $U(\widehat{\mathcal{P}}_1, p_1) \subseteq U_1$. Conversely, every *minimal pair* $(pw, \sigma^R) \in U_1$ (i.e. $\nexists \sigma' \prec \sigma$. $(pw, \sigma'^R) \in U_1$) is such that each state of σ enables the firing of a transition of $\mathcal{P}_1$ in a run from p_1 to pw. Therefore $p_1 \stackrel{\sigma}{\Longrightarrow}_{\widehat{\mathcal{P}}_1} pw$, which implies that $U_1 \subseteq U(\widehat{\mathcal{P}}_1, p_1)$.

Now, let $\widetilde{\mathcal{P}}_2 = (P_2, P_2, \Gamma_2, \Delta_2')$ be the LPDS such that $\langle p, u\rangle \stackrel{p'}{\hookrightarrow} \langle p', u'\rangle$ is in Δ_2' iff: (i) $\langle p, u\rangle \hookrightarrow \langle p', u'\rangle \in \Delta_2$ or (ii) $p = p'$ and $u = u' = \varepsilon$. We prove that $U_2 = D(\widetilde{\mathcal{P}}_2, p_2)$: Let $q_0 w_0 \Longrightarrow_{\mathcal{P}_2} q_1 w_1 \cdots \Longrightarrow_{\mathcal{P}_2} q_m w_m$ be a computation of $\mathcal{P}_2$ with $q_0 w_0 = p_2$ and $q_m w_m = p' w'$. Then, for every q_i (1) either q_i is observed several times by $\mathcal{P}_1$ (i.e., it is used to enable the firing of several transitions of $\mathcal{P}_1$), (2) or q_i is not observed at all by $\mathcal{P}_1$. Hence, the set of sequences of states that can be observed by $\mathcal{P}_1$ during this computation is $(q_0^+ q_1^+ \cdots q_m^+)\downarrow$. The sequences in $q_0^* q_1^+ \cdots q_m^+$ are traces of $\widetilde{\mathcal{P}}_2$, where the stuttering property is ensured by the rules (ii) that add loops on every state. Taking the downward closure gives the set we are looking for, i.e., $U_2 = D(\widetilde{\mathcal{P}}_2, p_2)$.

Then, the 2-tape automaton T can be computed as follows: (1) We construct a 2-tape automaton T_1 (resp. T_2) accepting $U(\widehat{\mathcal{P}}_1, p_1)$ (resp. $D(\widetilde{\mathcal{P}}_2, p_2)$). T_1 and T_2 are constructed following the proof of Theorem 1, (2) we compose T_1 and T_2 according to their second tape, and (3) we abstract away the second tape in the composed automaton. Formally, $T = \Pi_{(1,3)}(T_1 \circ_{(2,2)} T_2)$. We prove that $L(T) = post^*(C)$ [4].

The general case: Consider an APN_{obs} $N = (\mathcal{P}_1, \ldots, \mathcal{P}_n, R)$ of depth 1. The construction above can be extended to the general case as follows: Suppose that processes $\mathcal{P}_k, \ldots, \mathcal{P}_n$ are of depth 0 and $\mathcal{P}_1, \ldots, \mathcal{P}_{k-1}$ are of depth 1. Assume also w.l.o.g. that the processes of depth 1 can observe all the processes of depth 0 (if $\mathcal{P}_i$ does not observe $\mathcal{P}_j$, we add to the constraints of its rules all the states of $\mathcal{P}_j$. For each $\mathcal{P}_j$ of depth 1, we compute a 2-tape automaton that recognizes $U(\widehat{\mathcal{P}}_j, p_j)$ such that the first tape contains a reachable configuration $p_j w_j$ of $\mathcal{P}_j$, and the second one contains the sequence $\sigma \in (P_k \times \cdots \times P_n)^*$ of vectors of control states of the processes $\mathcal{P}_k, \ldots, \mathcal{P}_n$ that are needed by $\mathcal{P}_j$ to reach $p_j w_j$. Then, we compose all these automata by synchronizing them on their second tape to get a k-tape automaton $\widehat{T}$ that recognizes a vector $(p_1 w_1, \ldots, p_{k-1} w_{k-1}, \sigma)$ iff for $\forall j \in \{1, \ldots, k-1\}$, $(p_j w_j, \sigma) \in U(\widehat{\mathcal{P}}_j, p_j)$.

Let $\sigma = (q_k^1, \ldots, q_n^1) \cdots (q_k^m, \ldots, q_n^m)$, and let $s_i = q_i^1 \cdots q_i^m$ for every $i \in \{k, \ldots, n\}$. For the next step, we transform $\widehat{T}$ as a n-tape automaton by considering each vector $(p_1 w_1, \ldots, p_{k-1} w_{k-1}, \sigma)$ as $(p_1 w_1, \ldots, p_{k-1} w_{k-1}, s_k, \ldots, s_n)$.

Then, we compute for every $\mathcal{P}_i$ of depth 0 a 2-tape automaton $\widetilde{T}_i$ that recognizes $D(\widetilde{\mathcal{P}}_i, p_i)$. We synchronise all these automata with $\widehat{T}$ (the second tape of $\widetilde{T}_i$ gets synchronized with the component of $\widehat{T}$ that corresponds to the states of P_i). Then, we project on the components corresponding to the configurations. The obtained n-tape automaton accepts $(p_1 w_1, \ldots, p_n w_n)$ iff it is in the reachability set of N.

Theorem 2. *Let $N = (\mathcal{P}_1, \ldots, \mathcal{P}_n, R)$ be a APN_{obs} of depth 1, and let $C = \{(p_1, \ldots, p_n)\}$ be a configuration. It is possible to construct a n-tape automaton T such that $L(T) = post_N^*(C)$. The number of states of T is doubly exponential in $\sum_{i=1}^n |P_i| + |\Gamma_i|$.*

6 Solving the Reachability Problem for APN_{obs}

We consider in this section the reachability problem between two sets of configurations C_1 and C_2 for a linear APN_{obs} $N = (\mathcal{P}_1, \ldots, \mathcal{P}_n, R)$, i.e., checking whether $\exists c_2 \in C_2, \exists c_1 \in C_1$ s.t. $c_1 \Longrightarrow_N^* c_2$. Notice that, by Proposition 3, linearity is not a restriction. We prove the following fact by a reduction of PCP [4].

Theorem 3. *Given a rational set C and a configuration c, the problem of checking whether C is reachable from c is undecidable for APN_{obs}.*

We show hereafter that the reachability problem between two recognizable sets is decidable. W.l.o.g., we consider the problem for a pair of source and target configurations where all the stacks are empty.

Theorem 4. *The configuration reachability problem for a linear APN_{obs} $N = (\mathcal{P}_1, \ldots, \mathcal{P}_n, R)$ is decidable in $2(n-1)$-exponential time in $\sum_{i=0}^{n} |P_i| + |\Gamma_i|$.*

Intuitively, to check whether the configuration $(p'_1, \ldots, p'_n)$ is reachable from $(p_1, \ldots, p_n)$, we proceed inductively as follows: We start from index 1, and let A_1 be an automaton recognizing P_1^*. For every $i \in \{1, \ldots, n-1\}$, we construct an automaton $\mathcal{A}_{i+1}$ that recognizes the set of state sequences $\sigma \in P_{i+1}^*$ such that, if $\mathcal{P}_{i+1}$ has a run generating σ (modulo stuttering), then $\mathcal{P}_i$ has a computation from p_i to p'_i generating a sequence (modulo stuttering) in $L(\mathcal{A}_i)$. We consider stuttering since $\mathcal{P}_i$ (resp. $\mathcal{P}_{i-1}$) can observe several times $\mathcal{P}_{i+1}$ (resp. $\mathcal{P}_i$) in the same state. The set $L(\mathcal{A}_{i+1})$ is upward closed since if $\mathcal{P}_i$ requires observing the sequence σ of $\mathcal{P}_{i+1}$ for its computation, then $\mathcal{P}_i$ can perform the same computation if $\mathcal{P}_{i+1}$ generates (modulo stuttering) a sequence σ' such that $\sigma \preceq \sigma'$. We consider an LPDS $\widehat{\mathcal{P}_i}$ such that $Traces_{\widehat{\mathcal{P}_i}}(p_i, p'_i)$ is the set of observation sequences required by $\mathcal{P}_i$. Let $\widehat{\mathcal{P}_i} \otimes \mathcal{A}_i$ be the restriction of $\widehat{\mathcal{P}_i}$ to runs generating sequences in $L(\mathcal{A}_i)$ modulo stuttering. The automaton $\mathcal{A}_{i+1}$ can be obtained by constructing the upward closure of $Traces_{\widehat{\mathcal{P}_i} \otimes \mathcal{A}_i}(p_i, p'_i)$. Then, if $\mathcal{P}_n$ has a computation from p_n to p'_n generating a sequence in $L(\mathcal{A}_n)$, then $(p'_1, \ldots, p'_n)$ is reachable from $(p_1, \ldots, p_n)$ in N. Let us give the formal description of the decision procedure.

Definition 1. *For any $i \in \{1, \ldots, n-1\}$, let $\widehat{\mathcal{P}_i} = (P_i, P_{i+1}, \Gamma_i, \Delta'_i)$ be the LPDS s.t. $\langle p, u\rangle \overset{s}{\hookrightarrow} \langle p', u'\rangle \in \Delta'_i$ iff $\exists (\phi : \langle p, u\rangle \hookrightarrow \langle p', u'\rangle) \in \Delta_i$ such that $s \in \phi$, and let $\widehat{\mathcal{P}_n} = (P_n, P_n, \Gamma_n, \Delta'_n)$ be the LPDS s.t. $\langle p, u\rangle \overset{\varepsilon}{\hookrightarrow} \langle p', u'\rangle \in \Delta'_n$ iff $\langle p, u\rangle \hookrightarrow \langle p', u'\rangle \in \Delta_n$.*

Definition 2. *Given an LPDS $\mathcal{P} = (P, \Sigma, \Gamma, \Delta)$ and an automaton $\mathcal{A} = (A, P, \delta, I, F)$, let $\mathcal{P} \otimes \mathcal{A} = (P \times A, \Sigma, \Gamma, \Delta')$ be the LPDS where Δ' is the set of transition rules such that: (1) $\langle (p,s), \varepsilon\rangle \overset{\varepsilon}{\hookrightarrow} \langle (p,s'), \varepsilon\rangle \in \Delta'$ iff $(s, p, s') \in \delta$, and (2) $\langle (p,s), u\rangle \overset{a}{\hookrightarrow} \langle (p',s'), u'\rangle \in \Delta'$ iff $\langle p, u\rangle \overset{a}{\hookrightarrow} \langle p', u'\rangle \in \Delta$ and $(s, p', s') \in \delta$.*

Then, to solve the reachability problem between two configurations $(p_1, \ldots, p_n)$ and $(p'_1, \ldots, p'_n)$, we compute a sequence of automata $\mathcal{A}_i = (A_i, P_i, \delta_i, s_i, f_i)$, in the increasing ordering of their indices, where s_i is the initial state and f_i is the final state, defined as follows: (1) $\mathcal{A}_1$ recognizes P_1^* *(since there is no constraint on $\mathcal{P}_1$)*, and (2) $\forall i \in \{1, \ldots, n-1\}$, $\mathcal{A}_{i+1}$ recognizes the regular language $Traces_{\widehat{\mathcal{P}_i} \otimes \mathcal{A}_i}((p_i, s_i), (p'_i, f_i))\uparrow$.

Lemma 1. $(p_1,\ldots,p_n) \Longrightarrow_N^* (p'_1,\ldots,p'_n)$ *iff* $Traces_{\widehat{\mathcal{P}_n}\otimes\mathcal{A}_n}((p_n,s_n),(p'_n,f_n)) \neq \emptyset$.

Remark 3. The above algorithm is *top-down*: it starts from process $\mathcal{P}_1$ down to process $\mathcal{P}_n$. We can also use a *bottom-up* algorithm that starts from process $\mathcal{P}_n$ up to process $\mathcal{P}_1$ as follows: For every index i, we compute the set $\mathcal{A}'_i$ of state sequences that $\mathcal{P}_{i+1}$ can perform to go from p_{i+1} to p'_{i+1}. Then, we compute an LPDS that performs the same transitions as $\mathcal{P}_i$ when the sequences of states performed by $\mathcal{P}_{i+1}$ are in $\mathcal{A}'_i$ (remember that $\mathcal{P}_i$ observes $\mathcal{P}_{i+1}$). This LPDS is computed by a kind of product between $\mathcal{A}'_i$ and $\mathcal{P}_i$. $\mathcal{A}'_{i-1}$ can be computed as the downward closure of the language of the product between $\mathcal{P}_i$ and $\mathcal{A}'_i$. We start by computing $\mathcal{A}'_{n-1}$. Then $(p'_1,\ldots,p'_n)$ is reachable from $(p_1,\ldots,p_n)$ iff the pushdown process corresponding to the restriction of $\mathcal{P}_1$ can go from p_1 to p'_1. This bottom-up procedure is in $(n-1)$-exponential time in $\sum_{i=0}^{n}|P_i|+|\Gamma_i|$.

7 Acyclic Lossy Channel Pushdown Networks

An Acyclic Lossy Channel Pushdown Network (APN_{lc} for short) is a tuple $H = (\mathcal{P}_1,\ldots,\mathcal{P}_n,C,M)$ where: (1) $C \subseteq \{(j,i) : 1 \leq i < j \leq n\}$ is a finite set of unidirectional channels defining an acyclic graph, (2) M is a finite set of messages, and (3) $\forall i \in \{1,\ldots,n\}$, $\mathcal{P}_i = (P_i,\Sigma_i,\Gamma_i,\Delta_i)$ is a *communicating* pushdown system, where P_i a finite set of states, $\Sigma_i = (\{!\}\times M\times\{j : (i,j)\in C\})\cup(\{?\}\times M\times\{j : (j,i)\in C\})\cup(\{nop\})$ is a finite set of transition labels, Γ_i is a finite stack alphabet, and Δ_i is a finite set of transition rules of the form: $\langle p,u\rangle \overset{a}{\hookrightarrow} \langle p',u'\rangle$ where $a\in\Sigma_i$, $p,p'\in P_i$, and $u,u'\in\Gamma_i^*$ such that either (i) $|u|=1$ and $u'=\varepsilon$, (ii) $u=\varepsilon$ and $|u'|=1$, or (iii) $u=u'=\varepsilon$.

A transition of $\mathcal{P}_i$ labeled by $(!,m,j)$ means " $\mathcal{P}_i$ sends message m via the channel $(i,j)\in C$ to $\mathcal{P}_j$", whereas a transition labeled by $(?,m,j)$ means "$\mathcal{P}_i$ receives message m from the channel $(j,i)\in C$ sent by $\mathcal{P}_j$". A *nop* corresponds to an internal action.

An APN_{lc} is said to be *linear* if C is of the form $\{(i+1,i) : i\in\{1,\ldots,n-1\}\}$. The depth $d(i)$ of $\mathcal{P}_i$ is its depth in graph of the binary relation $R=\{(i,j) : (j,i)\in C\}$. The depth of H, $d(H)$ is $\max\{d(i) : 1\leq i\leq n\}$.

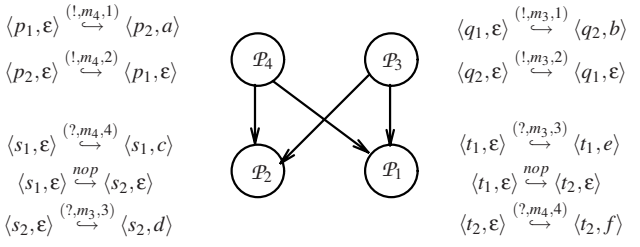
A *configuration* of H is a vector $\langle p_1w_1,\ldots,p_nw_n,\mathcal{V}\rangle$ where $p_iw_i\in P_i\Gamma_i^*$ is a local configuration of $\mathcal{P}_i$ and $\mathcal{V}: C\to M^*$ such that $\mathcal{V}(i,j)$ is the content of the channel (i,j). Let $\mathcal{E}$ be the mapping s.t. $\mathcal{E}(c)=\varepsilon$ for every $c\in C$.

We define a *transition relation* $\Longrightarrow_H$ between configurations as follows: $\langle p_1w_1,\ldots,p_nw_n,\mathcal{V}\rangle \Longrightarrow_H \langle p'_1w'_1,\ldots,p'_nw'_n,\mathcal{V}'\rangle$ iff $\exists i\in\{1,\ldots,n\}$ and $\exists(\langle p,u\rangle \overset{a}{\hookrightarrow} \langle p',u'\rangle)\in\Delta_i$ such that: (1) $p=p_i$ and $p'=p'_i$, (2) $w_i=uv$ and $w'_i=u'v$ for some $v\in\Gamma_i^*$, (3) $\forall j\neq i.\ p_j=p'_j$ and $w_j=w'_j$, and (4) either (i) $a=(?,m,k)$ is a *receive* operation; $m\mathcal{V}'(k,i)\preceq\mathcal{V}(k,i)$ and $\mathcal{V}'(j,l)\preceq\mathcal{V}(j,l)$ for every $(j,l)\in C$ s.t. $(j,l)\neq(k,i)$ (message m is read from the channel (k,i), and the contents of all the channels can lose some messages). Or (ii) $a=(!,m,k)$ is a *send* operation, and $\mathcal{V}'(i,k)\preceq\mathcal{V}(i,k)m$ and $\mathcal{V}'(j,l)\preceq\mathcal{V}(j,l)$ for every $(j,l)\in C$ s.t. $(j,l)\neq(i,k)$ (m is added to the channel (i,k) that receives the message and all the channels can lose messages). Or (iii) $a=nop$, and $\mathcal{V}'(j,l)\preceq\mathcal{V}(j,l)$ for every $(j,l)\in C$ (to express the loss of messages). Let $\Longrightarrow_H^*$ and $post_H^*$ denote respectively the reflexive-transitive closure of $\Longrightarrow_H$ and $post_H$.

Proposition 5. *The reachability problem for APN_{obs}'s can be reduced to the same problem for APN_{lc}'s. Conversely, the configuration reachability problem for linear APN_{lc}'s can be reduced to the same problem for linear APN_{obs}'s.*

8 Computing $post^*$ Images for $\mathbf{APN_{lc}}$

Consider the network $H_1 = (\mathcal{P}_1, \ldots, \mathcal{P}_4, \{(4,1),(3,1),(4,2),(3,2)\}, \{m_3, m_4\})$ given in the figure below with an initial configuration $\langle t_1, s_1, q_1, p_1, \mathcal{E}\rangle$.



Process $\mathcal{P}_4$ (resp. $\mathcal{P}_3$) loops on pushing an a (resp. b) in its stack while sending the same message m_4 (resp. m_3) to $\mathcal{P}_1$ and $\mathcal{P}_2$. During the execution of $\mathcal{P}_3$ and $\mathcal{P}_4$, $\mathcal{P}_2$ (resp. $\mathcal{P}_1$) cycles on pushing a c (resp. e) in its stack and removing a message m_4 (resp. m_3) from the channel $(4,2)$ (resp. $(3,1)$), then, it cycles on pushing a d (resp. f) in its stack while removing a message m_3 (resp. m_4) from the channel $(3,2)$ (resp. $(4,1)$). Hence, the set of reachable configurations when all channels are empty is $\{\langle t_2 f^i e^l, s_2 d^j c^k, q_1 b^m, p_1 a^n, \mathcal{E}\rangle : i,k \leq n \text{ and } l,j \leq m\}$ which is not rational.

Proposition 6. *Given an APN_{lc} H of depth 1, and a configuration c of H, the set $post^*_H(c)$ is not rational in general.*

We consider now the class of networks of depth 1 such that the undirected graph of the binary relation C is a forest (examples of such networks are given below).

Theorem 5. *Let $H = (\mathcal{P}_1, \ldots, \mathcal{P}_n, C, M)$ be an APN_{lc} of depth 1 such that the undirected graph of the binary relation C is a forest and let c be an initial configuration of H. Then, it is possible to construct a $(n+|C|)$-tape automaton T such that $L(T) = post^*_H(c)$. The number of states of T is doubly exponential in $\sum_{j=1}^{n} |P_j| + |\Gamma_j| + |M|$.*

We sketch hereafter the proof of the theorem above in the case where H has two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ and a channel $(2,1)$ (i.e., $H = (\mathcal{P}_1, \mathcal{P}_2, \{(2,1)\}, M)$). The extension to the general case is omitted here for lack of space. It can be found in [4].

W.l.o.g., we assume that in the initial configuration c, the stacks of $\mathcal{P}_1$ and $\mathcal{P}_2$ as well as the channel $(2,1)$ are empty. So, let $c = \langle p_1, p_2, \mathcal{E}\rangle$. We construct a 3-tape automaton T that accepts $post^*_H(c)$ as follows: First, we construct an LPDS $\breve{\mathcal{P}}_1 = (P_1, M, \Gamma_1, \Delta'_1)$

s.t. $p_1 \xRightarrow{\sigma}_{\breve{\mathcal{P}}_1} pw$ means that $\mathcal{P}_1$ reaches the configuration pw from p_1 provided that the sequence of messages consumed during the computation is σ. Δ_1' is defined as follows: (1) $\langle p,u\rangle \overset{m}{\hookrightarrow} \langle p',u'\rangle \in \Delta_1'$ iff $\langle p,u\rangle \overset{(?,m,2)}{\hookrightarrow} \langle p',u'\rangle \in \Delta_1$, and (2) $\langle p,u\rangle \overset{\varepsilon}{\hookrightarrow} \langle p',u'\rangle \in \Delta_1'$ iff $\langle p,u\rangle \overset{nop}{\hookrightarrow} \langle p',u'\rangle \in \Delta_1$.

Second, we construct an LPDS $\bar{\mathcal{P}}_2 = (P_2, M, \Gamma_2, \Delta_2')$ such that $p_2 \xRightarrow{\sigma}_{\bar{\mathcal{P}}_2} p'w'$ means that $\mathcal{P}_2$ reaches the configuration $p'w'$ from p_2 and the sequence of sent messages to the channel is σ. Δ_2' is defined as follows: (1) $\langle p,u\rangle \overset{m}{\hookrightarrow} \langle p',u'\rangle \in \Delta_2'$ iff $\langle p,u\rangle \overset{(!,m,1)}{\hookrightarrow} \langle p',u'\rangle \in \Delta_2$, and (2) $\langle p,u\rangle \overset{\varepsilon}{\hookrightarrow} \langle p',u'\rangle \in \Delta_2'$ iff $\langle p,u\rangle \overset{nop}{\hookrightarrow} \langle p',u'\rangle \in \Delta_2$.

The 3-tape automaton T must accept $(p'w', \sigma, pw) \in P_2\Gamma_2^* \times M^* \times P_1\Gamma_1^*$ iff $\exists \sigma_1, \sigma_2 \in M^*$ s.t. : (1) $p_1 \xRightarrow{\sigma_1}_{\breve{\mathcal{P}}_1} pw$, (2) $p_2 \xRightarrow{\sigma_2}_{\bar{\mathcal{P}}_2} p'w'$, and (3) $\sigma_1\sigma \preceq \sigma_2$ (σ_1 is the sequence of messages required by $\mathcal{P}_1$, σ_2 is the sequence of messages sent by $\mathcal{P}_2$, and σ is what remains in the channel after $\mathcal{P}_1$ reads σ_1 from σ_2). Then, to obtain T, we construct the 2-tape automata $\widehat{T}(\breve{\mathcal{P}}_1, p_1)$ and $\widetilde{T}(\bar{\mathcal{P}}_2, p_2)$ that accept respectively $U(\breve{\mathcal{P}}_1, p_1)$ and $D(\bar{\mathcal{P}}_2, p_2)$, and we compose these automata using the operator $\diamond$ on their second tape (corresponding to the channel content).

Lemma 2. *Let $T = \widetilde{T}(\bar{\mathcal{P}}_2, p_2) \diamond_{(2,2)} \widehat{T}(\breve{\mathcal{P}}_1, p_1)$. Then, the 3-tape automaton T accepts the vector $(p'w', \sigma, pw)$ iff $\langle pw, p'w', \mathcal{V}\rangle \in post_H^*(c)$ where $\mathcal{V}(2,1) = \sigma$.*

9 Computing the Channels Language for $\text{APN}_{|c}$

We show in this section that the reachability problem for the whole class of $\text{APN}_{|c}$ is decidable. Moreover, we show that the projection of the reachability set on the channels is an effectively recognizable set.

Let $H = (\mathcal{P}_1, \ldots, \mathcal{P}_n, C, M)$ be an $\text{APN}_{|c}$, and $c = \langle p_1, \ldots, p_n, \mathcal{V}_0\rangle$, where $\mathcal{V}_0(i,j) = \varepsilon$ for every channel $(i,j) \in C$, be the initial configuration where all channels and stacks are empty. Let $(p_1', \ldots, p_n') \in \prod_{i=1}^n P_i$ be a tuple of states of H. We define L_H as follows:

Definition 3. *L_H is the channels language such that $((u_{i,j})_{(i,j)\in C}) \in L_H$ iff $\exists (w_1, \ldots, w_n) \in \Gamma_1^* \times \cdots \times \Gamma_n^*$ s.t. $\langle p_1'w_1, \ldots, p_n'w_n, \mathcal{V}\rangle \in post_H^*(c)$ and $\mathcal{V}(i,j) = u_{i,j}$.*

Theorem 6. *The channel language L_H is effectively recognizable, and can be effectively defined by an m-union of products of finite-state automata with at most m states, where m is n-exponential in $\sum_{j=1}^n (|P_j| + |\Gamma_j|) + |M|$.*

We sketch the proof of the theorem assuming that H is linear. (See [4] for the extension to the general case.) For presentation matters, we consider first the case of a system with three processes $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{P}_3$. First, we construct the LPDS $\bar{\mathcal{P}}_3 = (P_3, M, \Gamma_3, \Delta_3')$ such that $p_3 \xRightarrow{\sigma}_{\bar{\mathcal{P}}_3} pw$ means that there is a run of $\mathcal{P}_3$ from pw to p_3, and the sent sequence of message to the channel along this run is σ. Then, let $\mathcal{A}_3 = (Q_3, M, \delta_3, s_3, f_3)$ be a finite-state automaton such that $L(\mathcal{A}_3)$ is the regular language $Traces_{\bar{\mathcal{P}}_3}(p_3, p_3'\Gamma_3^*)\downarrow$. In fact, $\mathcal{A}_3$ recognizes all possible contents of the channel $(3,2)$ when $\bar{\mathcal{P}}_3$ reaches $p_3'\Gamma_3^*$ from p_3. (We take the downward closure because the channels are lossy).

Then, we construct an automaton $\mathcal{A}_2$ that recognizes the set of all message sequences sent by $\mathcal{P}_2$ to the channel $(2,1)$ while consuming messages from the channel $(3,2)$, i.e.,

message sequences in $L(\mathcal{A}_3)$. For that, we compute a kind of product $\mathcal{P}_2 \star \mathcal{A}_3$ between $\mathcal{P}_2$ and $\mathcal{A}_3$ defined as follows:

Definition 4. *Given an index i and an automaton $\mathcal{A} = (Q, M, \delta, s, f)$, let $\mathcal{P}_i \star \mathcal{A} = (P_i \times Q, M, \Gamma_i, \Delta_i')$ be an LPDS where Δ_i' is defined as follows: (1) $\langle (p,q), u\rangle \overset{\varepsilon}{\hookrightarrow} \langle (p',q'), u'\rangle \in \Delta_i'$ iff $\langle p, u\rangle \overset{(?,m,i+1)}{\hookrightarrow} \langle p', u'\rangle \in \Delta_i$ and $(q, m, q') \in \delta$, (2) $\langle (p,q), u\rangle \overset{m}{\hookrightarrow} \langle (p',q'), u'\rangle \in \Delta_i'$ iff $\langle p, u\rangle \overset{(!,m,i-1)}{\hookrightarrow} \langle p', u'\rangle \in \Delta_i$ and $q = q'$, and (3) $\langle (p,q), u\rangle \overset{\varepsilon}{\hookrightarrow} \langle (p',q'), u'\rangle$ is in Δ_i' iff $\langle p, u\rangle \overset{nop}{\hookrightarrow} \langle p', u'\rangle \in \Delta_i$ and $q = q'$.*

The automaton $\mathcal{P}_2 \star \mathcal{A}_3$ behaves like $\mathcal{P}_2$ while consuming messages from $\mathcal{A}_3$ (due to rules (1)). Moreover, the language of $\mathcal{P}_2 \star \mathcal{A}_3$ is the set of message sequences sent by $\mathcal{P}_2$ to the channel $(2,1)$ (due to rules (2)). Since $\mathcal{P}_2$ consumes only a part (a prefix) of the content of the channel, for each state $q_3 \in Q_3$ in $\mathcal{A}_3$, $Traces_{\mathcal{P}_2 \star \mathcal{A}_3}((p_2, s_3), (p_2', q_3)\Gamma_2^*)$ represents the set of message sequences sent by $\mathcal{P}_2$ along a run from p_2 to $p_2' w$, for some $w \in \Gamma_2^*$, while consuming a prefix of a word in $L(\mathcal{A}_3)$ that leads to q_3. Then, the set of message sequences that are left in the channel $(3,2)$ is $L(\mathcal{A}_3^{q_3})$.

Let $\mathcal{A}_{2,q_3}$ be an automaton s.t. $L(\mathcal{A}_{2,q_3}) = Traces_{\mathcal{P}_2 \star \mathcal{A}_3}((p_2, s_3), (p_2', q_3)\Gamma_2^*)\downarrow$, i.e., $\mathcal{A}_{2,q_3}$ recognizes all possible contents of the channel $(2,1)$ when the content of the channel $(3,2)$ is in $L(\mathcal{A}_3^{q_3})$. Since $\mathcal{P}_1$ consumes messages from sequences in $L(\mathcal{A}_{2,q_3})$, for every state $q_2 \in Q_2$ in $\mathcal{A}_{2,q_3}$, we need to check whether there is a run of $\mathcal{P}_1$ from p_1 to $p_1' w$, for some $w \in \Gamma_1^*$, that consumes a prefix of a word recognized by a run of $\mathcal{A}_{2,q_3}$ ending at q_2, i.e., $Traces_{\mathcal{P}_1 \star \mathcal{A}_{2,q_3}}((p_1, s_2), (p_1', q_2)\Gamma_1^*)$ is empty. If not, $\mathcal{A}_{2,q_3}^{q_2} \times \mathcal{A}_3^{q_3}$ defines possible pairs of contents of the channels $(3,2)$ and $(2,1)$. Then, the contents of the channels is given by the union over all the pairs of states $q_2 \in Q_2$ and $q_3 \in Q_3$ of the automata $\mathcal{A}_{2,q_3}^{q_2} \times \mathcal{A}_3^{q_3}$ such that $Traces_{\mathcal{P}_1 \star \mathcal{A}_{2,q_3}}((p_1, s_2), (p_1', q_2)\Gamma_1^*) \neq \emptyset$.

For linear networks with $n \geq 2$ processes, we proceed as follows: First, we compute the finite-state automaton $\mathcal{A}_n = (Q_n, M, \delta_n, s_n, f_n)$ that recognizes $Traces_{\bar{\mathcal{P}}_n}(p_n, p_n' \Gamma_n^*)\downarrow$. Then, iteratively, for every $i \in \{n-1, \ldots, 2\}$ from $n-1$ down to 2, and for every $(q_n, \ldots, q_{i+1}) \in Q_n \times \cdots \times Q_{i+1}$, we compute the finite-state automaton $\mathcal{A}_{i,q_n,\ldots,q_{i+1}} = (Q_i, M, \delta_i, s_i, f_i)$ that recognizes $Traces_{\mathcal{P}_i \star \mathcal{A}_{i+1,q_n,\ldots,q_{i+2}}}((p_i, s_i), (p_i', q_{i+1})\Gamma_i^*)\downarrow$.

Lemma 3. $\forall (w_1, \ldots, w_n) \in \Gamma_1^* \times \cdots \times \Gamma_n^*$, $\langle p_1' w_1, \ldots, p_n' w_n, \mathcal{V}\rangle \in post_H^*(c)$ *iff* $\exists (q_n, \ldots, q_2) \in Q_n \times \cdots \times Q_2$ *s.t.* $Traces_{\mathcal{P}_1 \star A_{2,q_n,\ldots,q_3}}((p_1, s_2), (p_1', q_2)\Gamma_1^*) \neq \emptyset$ *and* $(\mathcal{V}(2,1), \ldots, \mathcal{V}(n-1,n)) \in L(\mathcal{A}_{2,q_n,\ldots,q_3}^{q_2} \times \mathcal{A}_{3,q_n,\ldots,q_4}^{q_3} \times \cdots \times \mathcal{A}_{n-1,q_n}^{q_{n-1}} \times \mathcal{A}_n^{q_n})$.

For linear networks, Theorem 6 is a consequence of Lemma 3. The construction above can be easily extended to the case of any acyclic network [4].

Finally, the previous algorithm can be adapted in order to prove the following fact.

Theorem 7. *The reachability problem between configurations (and therefore between recognizable sets) for APN_{lc} is decidable.*

10 Conclusion

We have proved the decidability of the reachability problem for acyclic networks of pushdown systems with communication mechanisms based on shared memory and

message-passing through lossy FIFO channels. Our models constitute an example of infinite-state systems for which the reachability problem is decidable, even though their reachability sets are not rational in general. In our work, we have explored the limits of recognizability/rationality of the reachability sets. We have shown that rationality is lost for depth 2, and for networks with lossy channels, it is lost even for depth 1 (unless the undirected graph of the network is a forest).

Our decidability proofs use automata constructions based on compositional analysis principles. A proof technique we use consists in analyzing the set of reachable configurations in a component of the network, assuming that its environment can provide some input (either an observable computation path, or a sequence of messages). On the other hand, we can analyze the reachable configurations together with the output sequences generated by the computations reaching these configurations. These two kinds of analysis allow to define respectively the input and the output interface of each pushdown process composing the network. Due to the communication mechanisms we consider, these interfaces (which are context-free languages) can be approximated without loss of preciseness (w.r.t. the considered reachability problem) by regular languages that are effectively computable (as upward and downward closures of context-free languages). Then, our proofs consist in showing that the input and output interfaces of a whole network can be "summarized" as a regular language. For that, (1) we isolate the extremal process, P_k say, (2) compute recursively the interface of the rest of the network, (3) compose this (regular) interface with P_k, which leads to a new pushdown system P'_k, and then (4) compute the interface of P'_k. This allows to reduce the reachability problem of our models to solving a sequence of decidable problems on single pushdown systems. (But obviously, this does not mean that our models can be reduced (or simulated) by a single pushdown system.) We believe that this kind of compositional analysis, based on computing upper/under approximate input and output interfaces, could be used for the (approximate) verification of pushdown networks, not only in the acyclic case.

Finally, a natural question which may raise is whether the reachability problem remains decidable if we allow switches between different acyclic communication relations. We prove that even for one of such a switch, the problem becomes undecidable for networks of depth (at least) 2 [4]. The case of networks of depth 1 is left open.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321 (1996)
2. Abdulla, P.A., Collomb-Annichini, A., Bouajjani, A., Jonsson, B.: Using forward reachability analysis for verification of lossy channel systems. FMSD 25(1), 39–65 (2004)
3. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inf. Comput. 127(2), 91–101 (1996)
4. Atig, M.F., Bouajjani, A., Touili, T.: On the reachability analysis of acyclic networks of pushdown systems. Research report, LIAFA lab (April 2008)
5. Berstel, J.: Transductions and context-free langages. TeubnerStudienbucher Informatik (1979)
6. Bouajjani, A., Esparza, J., Schwoon, S., Strejcek, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821. Springer, Heidelberg (2005)

7. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. 14(4), 551 (2003)
8. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653. Springer, Heidelberg (2005)
9. Bouajjani, A., Touili, T.: Reachability analysis of process rewrite systems. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 74–87. Springer, Heidelberg (2003)
10. Bouajjani, A., Touili, T.: On computing reachability sets of process rewrite systems. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467. Springer, Heidelberg (2005)
11. Bouajjani, A., Esparza, J.: Rewriting models of boolean programs. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 136–150. Springer, Heidelberg (2006)
12. Chadha, R., Viswanathan, M.: Decidability results for well-structured transition systems with auxiliary storage. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 136–150. Springer, Heidelberg (2007)
13. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing c programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
14. Chambard, P., Schnoebelen, P.: Post embedding problem is not primitive recursive, with applications to channel systems. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 265–276. Springer, Heidelberg (2007)
15. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural dataflow analysis. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578. Springer, Heidelberg (1999)
16. Esparza, J., Podelski, A.: Efficient algorithms for pre^* and $post^*$ on interprocedural parallel flow graphs. In: POPL, pp. 1–11 (2000)
17. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! TCS 256(1-2), 63–92 (2001)
18. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL. IEEE, Los Alamitos (2007)
19. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576. Springer, Heidelberg (2005)
20. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963. Springer, Heidelberg (2008)
21. Lammich, P., Müller-Olm, M.: Precise fixpoint-based analysis of programs with thread-creation and procedures. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703. Springer, Heidelberg (2007)
22. Lugiez, D., Schnoebelen, P.: The regular viewpoint on PA-processes. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 50–66. Springer, Heidelberg (1998)
23. Mayr, R.: Undecidable problems in unreliable computations. Theor. Comput. Sci. 1-3(297) (2003)
24. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440. Springer, Heidelberg (2005)
25. Schnoebelen, P.: Verifying lossy channel systems has nonprimitive recursive complexity. Inf. Process. Lett. 83(5) (2002)
26. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
27. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
28. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS, pp. 161–170. IEEE, Los Alamitos (2007)

Spatial and Behavioral Types in the Pi-Calculus*

Lucia Acciai and Michele Boreale

Dipartimento di Sistemi e Informatica
Università di Firenze
{lacciai,boreale}@dsi.unifi.it

Abstract. We present a framework that combines ideas from spatial logics and Igarashi and Kobayashi's behavioural type systems, drawing benefits from both. In our approach, type systems for the pi-calculus are introduced where newly declared (restricted) names are annotated with spatial process properties, predicating on those names, that are expected to hold in the scope of the declaration. Types are akin to ccs terms and account for the processes abstract behaviour and "shallow" spatial structure. The type systems relies on spatial model checking, but properties are checked against types rather than against processes. The considered class of properties is rather general and, differently from previous proposals, includes both safety and liveness ones, and is not limited to invariants.

Keywords: pi-calculus, behavioural type systems, spatial logic.

1 Introduction

In the past few years, *spatial logics* [7, 9] have emerged as promising tools for analyzing properties of systems described in process calculi. These logics aim at describing the spatial structure of processes. This makes them apt to express properties related to distribution and concurrency. An easy to grasp example is the race freedom property, stating that at any time, nowhere in the system there are two output actions ready on the same channel. The spectrum of properties that can be expressed by combination of simple spatial and behavioral connectives is very rich (see e.g. [7]). This richness is rather surprising, given the intensional nature of such logics: the process equivalences they induce coincide with, or come very close to, structural congruence (see e.g. [6]), a very fine equivalence that only permits elementary rearrangements of term structure.

A by known well-established trend in the field of process calculi is the use of *behavioural type systems* to simplify the analysis of concurrent message-passing programs [8, 10, 11]. Roughly, behavioural types are abstract representations of processes, yet sufficiently expressive to capture some properties of interest. In Igarashi and Kobayashi's work on generic type systems [10], the paper that pioneered this approach, processes are pi-calculus terms, while types are akin to simpler ccs terms. The crucial property enjoyed by the system is type soundness: in essence, for a certain class of properties (expressed in a simple modal logic), it holds that if a property is satisfied by a type then it is also satisfied by processes that inhabit that type. Results of this sort can in principle be used to effectively combine type checking and model checking. That is, in some cases

* Research partly supported by the EU within the FET-GC2 initiative, project Sensoria.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 372–386, 2008.

it is possible to replace (expensive) model checking on message-passing processes by (cheaper) model checking on types. The paper [8] further elaborates on these themes. A limitation of behavioural type systems proposed so far concerns the kind of properties that can be tackled this way. Essentially, in [8, 10], properties for which a general type soundness theorem works are safety invariants.

In the present paper, we try to combine the expressiveness of spatial logics with the effectiveness of behavioural type systems. More specifically, building on Igarashi and Kobayashi's work on generic type systems, we present type systems for the pi-calculus where newly declared (restricted) names are annotated with properties that predicate on those names. A process in the scope of a restriction is expected to satisfy, at run-time, the property expressed by the formula. We shall focus on properties expressible in a spatial logic – the *Shallow Logic* – which is a fragment of Caires and Cardelli's logic. Types are akin to ccs terms and account for (abstract) behaviour and "shallow" spatial structure of processes. The type system relies on (spatial) model checking: however, properties are checked against types rather than against processes. The considered class of properties is rather general and, unlike previous proposals [8, 10], includes both safety and liveness ones, and it is not limited to invariants. Several examples of such properties – including race freedom, deadlock freedom and many others – are given throughout the paper. As another contribution of the paper, we elaborate a distinction between *locally* and *globally checkable* properties. Informally, a locally checkable property is one that can be model-checked against any type by looking at the (local) names it predicates about, while hiding the others; a globally checkable one requires looking also at names causally related to the local ones, hence in principle at names declared elsewhere in the process. These two classes of properties correspond in fact to two distinct type systems, exhibiting different degrees of compositionality and effectiveness (with the global one less compositional/effective). To sum up, we make the following contributions:

- we establish an explicit connection between spatial logics and behavioural type systems. In this respect, a key observation is that processes and the behavioural types they inhabit share the same "shallow" spatial structure, which allows us to prove quite precise correspondences between them and general type soundness theorems;
- we syntactically identify classes of formulae for which type soundness is guaranteed;
- unlike previous proposals, our type soundness results are not limited to safety properties nor to invariant properties;
- we investigate a distinction between locally and and globally checkable properties.

Structure of the paper. In Section 2 we introduce the language of processes, a standard polyadic pi-calculus. In Section 3 we introduce both spatial properties and the Shallow Logic, a simple language to denote them. In Section 4 the first type system, tailored to "local" properties, is presented and thoroughly discussed. Type soundness for this system is then discussed in Section 5 along with a few examples. A "global" variant of the type system, a soundness result and a few examples are presented and discussed in Section 6. Some limitations of our approach, and possible workarounds for them, are discussed in Section 7. A few remarks on further and related work conclude the paper in Section 8. Proofs are omitted due to space limitations.

2 A Process Calculus

Processes. The language we consider is a synchronous polyadic pi-calculus [14] with guarded summations and replications. As usual, we presuppose a countable set $\mathcal{N}$ of *names*. We let lowercase letters $a, b, ..., x, y, ...$ range over names, and $\tilde{a}, \tilde{b}, ...$ range over tuples of names. Processes $P, Q, R, \ldots$ are defined by the grammar below

$$\alpha ::= a(\tilde{b}) \mid \overline{a}\langle\tilde{b}\rangle \mid \tau \qquad P ::= \textstyle\sum_{i\in I} \alpha_i.P_i \mid P|P \mid (\nu\tilde{b} : \tilde{\mathsf{t}})P \mid\ !a(\tilde{b}).P\,.$$

In the restriction clause, $\tilde{\mathsf{t}}$ is a tuple of channel types, to be defined later in Subsection 2, such that $|\tilde{\mathsf{t}}| = |\tilde{b}|$. Note that restriction acts on tuples $\tilde{b} = b_1, ..., b_n$, rather than on individual names. Indeed, the form $\nu\tilde{b}$ is equivalent to $\nu b_1 \cdots \nu b_n$ from an operational point of view. From the point of view of the type systems, however, the form $\nu\tilde{b}$ will allow us to specify properties that should hold of a group of names, as we shall see in later section. Notions of free names $\mathrm{fn}(\cdot)$, of bound names and of alpha-equivalence arise as expected, and terms are identified up to alpha-equivalence. In particular, we let $\mathrm{fn}((\nu\tilde{b} : \tilde{\mathsf{t}})P) = (\mathrm{fn}(P) \cup \mathrm{fn}(\tilde{\mathsf{t}})) \setminus \tilde{b}$. To avoid arity mismatches in communications, we shall only consider terms that are well-sorted in some fixed sorting system (see e.g. [14]), and call $\mathcal{P}$ the resulting set of *processes*.

Notation. We shall write $\mathbf{0}$ for the empty summation. Trailing $\mathbf{0}$'s will be often omitted. Given $n \geq 0$ tuples of names $\tilde{b}_1, ..., \tilde{b}_n$, we abbreviate $(\nu\tilde{b}_1 : \tilde{\mathsf{t}}_1) \cdots (\nu\tilde{b}_n : \tilde{\mathsf{t}}_n)P$ as $(\tilde{\nu}\,\tilde{b}_i : \tilde{\mathsf{t}}_i)_{i\in 1..n}P$, or simply $(\tilde{\nu}\tilde{b})P$ when no ambiguity about the $\tilde{\mathsf{t}}_i$ arises. In general, channel types annotations may be omitted when not relevant for the discussion. For any tuple/set of names $\tilde{x}$ and item t, $\tilde{x}\#t$ means that $\tilde{x} \cap \mathrm{fn}(t) = \emptyset$. This is extended to tuples of items $\tilde{t}$, written $\tilde{x}\#\tilde{t}$, as expected.

Over $\mathcal{P}$, we define a *reduction semantics*, based as usual on a notion of structural congruence and on a (labelled) reduction relation. These relations are defined, respectively, as the least congruence $\equiv$ and as the least relation $\xrightarrow{\lambda}$ generated by the axioms in Table 1 and Table 2. As usual, the structural law for replication is replaced by a suitable reduction rule. Concerning Table 1, note that, similarly to [10], we drop two laws for restrictions ($(\nu\tilde{y})\mathbf{0} = \mathbf{0}$ and $(\nu\tilde{x})(\nu\tilde{y})P = (\nu\tilde{y})(\nu\tilde{x})P$): these laws become problematic once restrictions will be decorated with formulae containing free names. Concerning Table 2, note that we annotate each reduction with a label λ that carries information on the (free) subject name involved in the corresponding synchronization if any: $\lambda ::= \langle a\rangle \mid \langle\epsilon\rangle$. We define a hiding operator on labels, written $\lambda \uparrow_{\tilde{b}}$, as follows: $\lambda \uparrow_{\tilde{b}} = \langle a\rangle$ if $\lambda = \langle a\rangle$ and $a \notin \tilde{b}$, $\lambda \uparrow_{\tilde{b}} = \epsilon$ otherwise.

Notation. In the sequel, for $\sigma = \lambda_1 \cdot \cdots \cdot \lambda_n$, $P \xrightarrow{\sigma} Q$ means $P \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} Q$, and $P \to Q$ (resp. $P \to^* Q$) means $P \xrightarrow{\lambda} Q$ (resp. $P \xrightarrow{\sigma} Q$) for some λ (resp. σ). Moreover,

Table 1. Laws for structural congruence $\equiv$ on processes

$$P|\mathbf{0} \equiv P \qquad (P|Q)|R \equiv P|(Q|R) \qquad P|Q \equiv Q|P \qquad (\nu\tilde{x} : \tilde{\mathsf{t}})P|Q \equiv (\nu\tilde{x} : \tilde{\mathsf{t}})(P|Q) \quad \text{if } \tilde{x}\#Q$$

Table 2. Rules for the reduction relation $\xrightarrow{\lambda}$ on processes

$$(\text{COM})\ \frac{\alpha_l = a(\tilde{x}) \quad \beta_n = \overline{a}\langle\tilde{b}\rangle \quad l \in I \quad n \in J}{\sum_{i\in I}\alpha_i.P_i \mid \sum_{j\in J}\beta_j.Q_j \xrightarrow{\langle a\rangle} P_l[\tilde{b}/\tilde{x}] \mid Q_n} \qquad (\text{TAU})\ \frac{j \in I \quad \alpha_j = \tau}{\sum_{i\in I}\alpha_i.P_i \xrightarrow{\langle\epsilon\rangle} P_j}$$

$$(\text{REP-COM})\ \frac{\beta_n = \overline{a}\langle\tilde{b}\rangle \quad n \in J}{!a(\tilde{x}).P \mid \sum_{j\in J}\beta_j.Q_j \xrightarrow{\langle a\rangle} !a(\tilde{x}).P \mid P[\tilde{b}/\tilde{x}] \mid Q_n} \qquad (\text{PAR})\ \frac{P \xrightarrow{\lambda} P'}{P \mid Q \xrightarrow{\lambda} P' \mid Q}$$

$$(\text{STRUCT})\ \frac{P \equiv Q \quad Q \xrightarrow{\lambda} Q' \quad Q' \equiv P'}{P \xrightarrow{\lambda} P'} \qquad (\text{RES})\ \frac{P \xrightarrow{\lambda} P'}{(\nu\tilde{x}:\tilde{\mathsf{t}})P \xrightarrow{\lambda\uparrow_{\tilde{x}}} (\nu\tilde{x}:\tilde{\mathsf{t}})P'}$$

we say that a process P has a *barb* a (resp. $\overline{a}$), written $P \searrow_a$ (resp. $P \searrow_{\overline{a}}$), if $P \equiv (\tilde{\nu b})(\sum_i \alpha_i.P_i + a(\tilde{x}).Q|R)$ (resp. $P \equiv (\tilde{\nu b})(\sum_i \alpha_i.P_i + \overline{a}\langle\tilde{c}\rangle.Q|R)$), with $a \notin \tilde{b}$.

Types. The set $\mathcal{T}$ of *types* $\mathsf{T}, \mathsf{S}, \mathsf{U}, \ldots$ is generated by the following grammar:

$$\mu ::= a(\mathsf{t}) \mid \overline{a} \mid \tau \qquad \mathsf{t} ::= (\tilde{x}:\tilde{\mathsf{t}})\mathsf{T} \qquad \mathsf{T} ::= \textstyle\sum_i \mu_i.\mathsf{T}_i \mid \mathsf{T}|\mathsf{T} \mid\ !a(\mathsf{t}).\mathsf{T} \mid (\nu\tilde{a}:\tilde{\mathsf{t}})\mathsf{T}$$

with $\tilde{x}\#\tilde{\mathsf{t}}$ and $\tilde{x} \subseteq \text{fn}(\mathsf{T})$. In channel types $(\tilde{x}:\tilde{\mathsf{t}})\mathsf{T}$, we stipulate that $(\tilde{x}:\tilde{\mathsf{t}})$ is a binder with scope T. Informally, $a(\mathsf{t}).\mathsf{T}$ is a process type where a can transport names of channel-type t. In a channel type $(\tilde{x}:\tilde{\mathsf{t}})\mathsf{T}$, $\tilde{x}$ and $\tilde{\mathsf{t}}$ represent, respectively, the formal parameters and types of objects that can be passed along the channel, while type T is a process type prescribing a usage of those parameters. Note that, in $(\tilde{x}:\tilde{\mathsf{t}})\mathsf{T}$, it might in general be $\text{fn}(\mathsf{T}) \setminus \tilde{x} \neq \emptyset$. In the sequel, we shall often omit writing the channel type $()\mathbf{0}$, writing e.g. $(x)\overline{x}$ instead of $(x:()\mathbf{0})\overline{x}$. Process types are akin to ccs terms bearing annotations on input prefixes and restrictions. Notions of free names, alpha-equivalence, structural congruence and reduction for types parallel those of processes. Note, that annotations contribute to the set of free names[1] of a type, but do not play a direct role in its reduction semantics (e.g. $\overline{c}.\mathsf{T}|c(\mathsf{t}).\mathsf{S} \xrightarrow{\langle c\rangle} \mathsf{T}|\mathsf{S}$).

3 Properties

We first take a general view of properties as *P-sets*, that is sets of processes and types (subject to certain conditions). Then introduce *Shallow Logic*, a simple language to denote a class of such properties. Although processes and types live in different worlds, for the purposes of this section it is possible and convenient to deal with them in a uniform manner. In what follows, we let $A, B, \ldots$ range over the set $\mathcal{U} \triangleq \mathcal{P} \cup \mathcal{T}$. Elements of $\mathcal{U}$ will be generically referred to as *terms*.

[1] Indeed, the reason for introducing these annotations is precisely to ensure that, in the type systems we shall introduce, whenever $\Gamma \vdash P : \mathsf{T}$ then $\text{fn}(P) \subseteq \text{fn}(\mathsf{T})$.

P-sets. Following [6, 7], a property set, P-set in brief, is a set of terms closed under structural congruence and having a finite support: this intuitively means that the set of names that are "relevant" for the property is finite (somewhat analogous to the notion of free names for syntactic terms). In the following, we let $\{a \leftrightarrow b\}$ denote the *transposition* of a and b, that is, the substitution that assigns a to b and b to a, and leave the other names unchanged. For $\Phi \subseteq \mathcal{U}$, we let $\Phi\{a \leftrightarrow b\}$ denote $\{A\{a \leftrightarrow b\} | A \in \Phi\}$.

Definition 1 (support, P-set, least support). *Let be $\Phi \subseteq \mathcal{U}$ and $N \subseteq \mathcal{N}$. (a) N is a* support *of Φ if for each $a, b \notin N$, it holds that $\Phi\{a \leftrightarrow b\} = \Phi$. (b) A* property set *(*P-set*) is a set of terms $\Phi \subseteq \mathcal{U}$ that is closed under $\equiv$ and has finite support. (c) The* least support *of Φ, written* $\mathrm{supp}(\Phi)$*, is defined as* $\mathrm{supp}(\Phi) \triangleq \bigcap_{N \text{ support of } \Phi} N$.

In other words, N is a support of Φ if renaming names *outside* N with fresh names does not affect Φ. P-sets have finite supports, and since countable intersection of supports is still a support, they also have a least support. In the rest of the paper we will deal with properties that need not be invariant through reductions. This calls for a notion of λ-*derivative* of a P-set Φ, describing the set of terms reachable via λ-reductions from terms in Φ: $\Phi_\lambda \triangleq \{B | \exists A \in \Phi : A \xrightarrow{\lambda} B\}$. The λ-derivative of a P-set is a P-set.

Proposition 1. *Let Φ be a P-set and λ be a reduction label. Then Φ_λ is a P-set and* $\mathrm{supp}(\Phi_\lambda) \subseteq \mathrm{supp}(\Phi)$.

The Ok predicate defined below individuates P-sets that enjoy certain desirable conditions. (1) requires a P-set to be closed under parallel composition with terms not containing free names (2) demands a P-set to be invariant under reductions that do not involve names in its support. Finally, (3) requires preservation of (1) and (2) under derivatives.

Definition 2 ($\mathrm{Ok}(\cdot)$ **predicate).** *We define* $\mathrm{Ok}(\cdot)$ *as the largest predicate on P-sets such that whenever* $\mathrm{Ok}(\Phi)$ *then: (1) for any $A, B \in \mathcal{P}$ s.t.* $\mathrm{fn}(B) = \emptyset$*: $A \in \Phi$ if and only if $A|B \in \Phi$; similarly for $A, B \in \mathcal{T}$; (2) $\Phi_\lambda = \Phi$ for $\lambda = \langle \epsilon \rangle$ or $\lambda = \langle b \rangle$ with $b \notin \mathrm{supp}(\Phi)$; (3) for each λ,* $\mathrm{Ok}(\Phi_\lambda)$ *holds.*

In the rest of the paper, we shall focus on properties represented by Ok P-sets.

Shallow Logic. The logic for the pi-calculus we introduce below can be regarded as a fragment of Caires and Cardelli's Spatial Logic [7]. We christen this fragment *Shallow Logic*, as it allows us to speak about the dynamic as well as the "shallow" spatial structure of processes and types. In particular, the logic does not provide for modalities that allows one to "look underneath" prefixes. Another important feature of this fragment is that the basic modalities focus on channel *subjects*, ignoring the object part at all. This selection of operators is sufficient to express a variety of interesting process properties (race freedom, unique receptiveness [16], deadlock freedom, to mention a few), while being tractable from the point of view of verification (see also Caires' [6]).

Definition 3 (Shallow Logic). *The set $\mathcal{F}$ of* Shallow Logic *formulae $\phi, \psi, \ldots$ is given by the following syntax, where $a \in \mathcal{N}$ and $\tilde{a} \subseteq \mathcal{N}$:*

$$\phi ::= \mathbf{T} \mid \phi \vee \phi \mid \langle a \rangle \phi \mid \langle \tilde{a} \rangle^* \phi \mid \langle -\tilde{a} \rangle^* \phi \mid \neg \phi \mid a \mid \overline{a} \mid \phi | \phi \mid \mathrm{H}^* \phi.$$

Table 3. Interpretation of formulae over terms

$$[[\mathbf{T}]] = \mathcal{U} \qquad [[\mathrm{H}^*\phi]] = \{A \mid \exists \tilde{a}, B : A \equiv (\tilde{\nu}\tilde{a})B, \tilde{a}\#\phi,\ B \in [[\phi]]\}$$

$$[[\phi_1 \vee \phi_2]] = [[\phi_1]] \cup [[\phi_2]] \qquad [[\phi_1|\phi_2]] = \{A \mid \exists A_1, A_2 : P \equiv A_1|A_2,\ A_1 \in [[\phi_1]],\ A_2 \in [[\phi_2]]\}$$

$$[[\neg\phi]] = \mathcal{U} \setminus [[\phi]] \qquad [[\langle a\rangle\phi]] = \{A \mid \exists B : A \xrightarrow{\langle a\rangle} B,\ B \in [[\phi]]\}$$

$$[[a]] = \{A \mid A \searrow_a\} \qquad [[\langle\tilde{a}\rangle^*\phi]] = \{A \mid \exists \sigma, B : P \xrightarrow{\sigma} B,\ \sigma \in \{\langle b\rangle | b \in \tilde{a}\}^*,\ B \in [[\phi]]\}$$

$$[[\overline{a}]] = \{A \mid A \searrow_{\overline{a}}\} \qquad [[\langle -\tilde{a}\rangle^*\phi]] = \{A \mid \exists \sigma, B : A \xrightarrow{\sigma} B,\ \tilde{a}\#\sigma,\ B \in [[\phi]]\}$$

The free names of a formula ϕ, written $\mathrm{fn}(\phi)$, are defined as expected. We let $\mathcal{F}_{\tilde{x}} = \{\phi \in \mathcal{F} : \mathrm{fn}(\phi) \subseteq \tilde{x}\}$. The set of logical operators includes spatial $(a, \overline{a}, |, \mathrm{H}^*)$ as well as dynamic $(\langle a\rangle, \langle\tilde{a}\rangle^*, \langle -\tilde{a}\rangle^*)$ connectives, beside the usual boolean connectives, including a constant $\mathbf{T}$ for "true". The interpretation of $\mathcal{F}$ over the set of processes is given in Table 3. Connectives are interpreted in the standard manner. We write $A \models \phi$ if $A \in [[\phi]]$.

Interpretations of formulae are P-sets, as stated below.

Lemma 1. *Let $\phi \in \mathcal{F}$. Then $[[\phi]]$ is a P-set and $fn(\phi) \supseteq \mathrm{supp}([[\phi]])$.*

Notation. In what follows, when no confusion arises, we shall often denote $\Phi = [[\phi]]$ just as ϕ. Moreover, we shall write $A \models \Phi$ to mean $A \in \Phi$. We abbreviate $\neg\langle -\tilde{x}\rangle^*\neg\phi$ as $\Box^*_{-\tilde{x}}\phi$. Moreover, $\langle -\emptyset\rangle^*\phi$ and $\Box^*_{-\emptyset}\phi$ are abbreviated as $\Diamond^*\phi$ and $\Box^*\phi$, respectively. Note that $\Diamond^*$ and $\Box^*$ correspond to the standard "eventually" and "always" modalities as definable, e.g., in the mu-calculus.

A further motivation for our particular selection of modalities is that satisfaction of any formula of $\mathcal{F}$ is, so to speak, invariant under parallel composition. In particular, whether A satisfies or not a property ϕ of a bunch of names $\tilde{x}$, should not depend on the presence of a parallel closed context B. Formulae of Cardelli and Caires' Spatial Logic outside $\mathcal{F}$ do not, in general, meet this requirement. As an example, the requirement obviously fails for $\neg(\neg\mathbf{0}|\neg\mathbf{0})$, saying that there is at most one non-null thread in the process. As another example, take the formula $\Diamond\mathbf{T}$, where $\Diamond$ is the one-step modality, saying that one reduction is possible: the reduction might be provided by the context B and not by A. This explains the omission of the one-step modality from Shallow Logic.

Lemma 2. *Let A be a term and $\phi \in \mathcal{F}_{\tilde{x}}$. For any term B such that $A|B$ is a term and $\mathrm{fn}(B) = \emptyset$ we have that $A \models \phi$ if and only $A|B \models \phi$.*

Example 1 (sample formulae). The following formulae define properties depending on a generic channel name a. They will be reconsidered several time throughout the paper.

$$\text{Race freedom:} \quad NoRace(a) \triangleq \Box^*\ \neg\mathrm{H}^*(\overline{a}|\overline{a})$$

$$\text{Unique receptiveness:} \quad UniRec(a) \triangleq \Box^*\ (a \wedge \neg\mathrm{H}^*(a|a))$$

$$\text{Responsiveness:} \quad Resp(a) \triangleq \Box^*_{-a}\Diamond^*\langle a\rangle$$

$$\text{Deadlock freedom:} \quad DeadFree(a) \triangleq \Box^*[\ (\overline{a} \to \mathrm{H}^*(\overline{a}|\Diamond^* a)) \wedge (a \to \mathrm{H}^*(a|\Diamond^* \overline{a}))\ ].$$

NoRace(a) says that it will never be the case that there are two concurrent outputs competing for synchronization on a. *UniRec*(a) says that there will always be exactly one

receiver ready on channel a. $Resp(a)$ says that, until a reduction on a does not take place, it is possible to reach a reduction on a. If a is a return channel of some invoked service or function, this means the service or function will, under a suitable fairness assumption, eventually respond (see also [3]). Finally, $DeadFree(a)$ says that each output on a will eventually meet a synchronizing input, and vice-versa.

We shall sometimes need to be careful about the placement of the modality $\langle -\tilde{a}\rangle^*$ with respect to negation $\neg$. To this purpose, it is convenient to introduce two subsets of formulae, positive and negative ones.

Definition 4 (positive and negative formulae). *We say a formula ϕ is* positive *(resp.* negative*) if each occurrence of modality $\langle -\tilde{a}\rangle^*$ in ϕ is in the scope of an even (resp. odd) number of negations "$\neg$".*

We let $\mathcal{F}^+$ (resp. $\mathcal{F}^-$) denote the subset of positive (resp. negative) formulae in $\mathcal{F}$. The sets $\mathcal{F}^+_{\tilde{x}}$ and $\mathcal{F}^-_{\tilde{x}}$ are defined as expected.

Example 2. Concerning the formulae introduced in Example 1, note that $NoRace(a)$ and $UniRec(a)$ are negative, while both $Resp(a)$ and $DeadFree(a)$ are neither positive nor negative, as in both the modality $\Diamond^*$ occurs both in negative and in positive position.

Note that our definitions of "positive" and "negative" are more liberal than the ones considered by Igarashi and Kobayashi [10], where the position of all spatial modalities – including the analogs of $|$, a and $\overline{a}$ – w.r.t. negation must be taken into account (e.g., unique receptiveness would not be considered as negative in the classification of [10]). This difference will have influential consequences on the generality of the type soundness theorems of the type systems. In the rest of the paper, we shall mainly focus on formulae whose denotations are Ok P-sets. We write $\mathrm{Ok}(\phi)$ if $\mathrm{Ok}([\![\phi]\!])$ holds. The following lemma provides a sufficient syntactic condition for a formula to be Ok.

Lemma 3. *Let ϕ be a Shallow Logic formula of the form either $\Box^*\psi$ or $\Box^*_{-\tilde{a}}\Diamond^*\psi'$, where ψ' does not contain $\neg$. Then* $\mathrm{Ok}(\phi)$.

Example 3. Formulas in Example 1 are in the format of Lemma 3, hence they are Ok.

4 A "Local" Type System

We present here our first type system. The adjective "local" refers to the controlled way P-set membership (that is, model checking, in practical cases) is checked.

Annotated processes. As anticipated in Section 2, the type system works on annotated processes. Each restriction introduces a property, under the form of an Ok P-set, that depends on the restricted names and is expected to be satisfied by the process in the restriction's scope. This means that, for annotated processes, the clause of restriction is modified thus $P ::= \cdots \mid (\nu\tilde{a} : \tilde{\mathrm{t}}\,; \Phi)P$ with $\tilde{a} \supseteq \mathrm{supp}(\Phi)$ and $\mathrm{Ok}(\Phi)$. For brevity, when no confusion arises we shall omit writing explicitly channel types and properties

in restrictions, especially when $\mathsf{t} = ()\mathbf{0}$ and $\Phi = [[\mathbf{T}]]$. The reduction rule for restriction on annotated processes takes into account the λ-derivative of Φ in the continuation process:

$$(\text{RES}) \quad \frac{P \xrightarrow{\lambda} P'}{(\nu\tilde{x} : \tilde{\mathsf{t}}\,; \Phi)P \xrightarrow{\lambda\uparrow_{\tilde{x}}} (\nu\tilde{x} : \tilde{\mathsf{t}}\,; \Phi_\lambda)P'} .$$

For an annotated process P, we take $P \models \phi$ to mean that the plain process obtained by erasing all annotations from P satisfies ϕ. A "good" process is one that satisfies its own annotations at an active position. Formally:

Definition 5 (well-annotated processes). *A process $P \in \mathcal{P}$ is* well-annotated *if whenever $P \equiv (\tilde{\nu b})(\nu\tilde{a} : \Phi)Q$ then $Q \models \Phi$.*

Typing rules. Judgements of type system are of the form $\Gamma \vdash P : \mathsf{T}$, where: $P \in \mathcal{P}$, $\mathsf{T} \in \mathcal{T}$ and Γ is a *context*, that is, a finite partial map from names $a, b, c, \ldots$ to channel types $\mathsf{t}, \mathsf{t}', \ldots$. We write $\Gamma \vdash a : \mathsf{t}$ if $a \in \mathrm{dom}(\Gamma)$ and $\Gamma(a) = \mathsf{t}$. We say that a context is *well-formed* if whenever $\Gamma \vdash a : (\tilde{x} : \tilde{\mathsf{t}})\mathsf{T}$ then $\mathrm{fn}(\mathsf{T}, \tilde{\mathsf{t}}) \subseteq \tilde{x} \cup \mathrm{dom}(\Gamma)$. In what follows we shall only consider well-formed contexts. Contexts are assumed to be well-formed in rules of the type system. In the type system, we make use of a "hiding" operation on types, $\mathsf{T} \downarrow_{\tilde{x}}$, which masks the use of names not in $\tilde{x}$ in T (as usual, in the definition we assume that all bound names in T and t are distinct from each other and disjoint from the set of free names and from $\tilde{x}$).

Definition 6 (hiding on types). *For any type T and $\tilde{x}$, we let $\mathsf{T} \downarrow_{\tilde{x}}$ denote the type obtained by replacing every occurrence of prefixes $a(\mathsf{t})$. and $\overline{a}$. with τ., for each $a \in \mathrm{fn}(\mathsf{T}) \setminus \tilde{x}$. Hiding on channel types, $\mathsf{t} \downarrow_{\tilde{x}}$, is defined similarly.*

E.g., $(\nu a : \mathsf{t})(a(\mathsf{t}).b(\mathsf{t}')|a(\mathsf{t}).c|\overline{c}|\overline{a}) \downarrow_b = (\nu a : \mathsf{t} \downarrow_{a,b})(a(\mathsf{t} \downarrow_{a,b}).b(\mathsf{t}' \downarrow_{a,b})|a(\mathsf{t} \downarrow_{a,b}).\tau|\tau|\overline{a})$. The rules of the type system are shown in Table 4. The structure of the system is along the lines of [10]; the main differences are discussed in Section 7. The key rules are (T-Inp), (T-Out), (T-Res) and (T-Eq). By and large, the system works as follows: given a process P, it computes an abstraction of P in the form of a type T. At any restriction $(\nu\tilde{a} : \tilde{\mathsf{t}}\,; \Phi)P$ (rule (T-Res)), the abstraction T obtained for P is used to check that P's usage of names

Table 4. Typing rules

$$(\text{T-Inp}) \quad \frac{\Gamma \vdash a : (\tilde{x} : \tilde{\mathsf{t}})\mathsf{T} \quad \Gamma, \tilde{x} : \tilde{\mathsf{t}} \vdash P : \mathsf{T}|\mathsf{T}' \quad \tilde{x} \# \mathsf{T}'}{\Gamma \vdash a(\tilde{x}).P : a((\tilde{x} : \tilde{\mathsf{t}})\mathsf{T}).\mathsf{T}'} \qquad (\text{T-Tau}) \quad \frac{\Gamma \vdash P : \mathsf{T}}{\Gamma \vdash \tau.P : \tau.\mathsf{T}}$$

$$(\text{T-Out}) \quad \frac{\Gamma \vdash a : (\tilde{x} : \tilde{\mathsf{t}})\mathsf{T} \quad \Gamma \vdash \tilde{b} : \tilde{\mathsf{t}} \quad \Gamma \vdash P : \mathsf{S}}{\Gamma \vdash \overline{a}\langle\tilde{b}\rangle.P : \overline{a}.(\mathsf{T}[\tilde{b}/\tilde{x}]\,|\,\mathsf{S})} \qquad (\text{T-Eq}) \quad \frac{\Gamma \vdash P : \mathsf{T} \quad \mathsf{T} \equiv \mathsf{S}}{\Gamma \vdash P : \mathsf{S}}$$

$$(\text{T-Sum}) \quad \frac{|I| \neq 1 \quad \forall i \in I : \ \Gamma \vdash \alpha_i.P_i : \mu_i.\mathsf{T}_i}{\Gamma \vdash \sum_i \alpha_i.P_i : \sum_i \mu_i.\mathsf{T}_i} \qquad (\text{T-Rep}) \quad \frac{\Gamma \vdash a(\tilde{x}).P : a(\mathsf{t}).\mathsf{T}}{\Gamma \vdash\, !a(\tilde{x}).P :\, !a(\mathsf{t}).\mathsf{T}}$$

$$(\text{T-Res}) \quad \frac{\Gamma, \tilde{a} : \tilde{\mathsf{t}} \vdash P : \mathsf{T} \quad \mathsf{T} \downarrow_{\tilde{a}} \models \Phi}{\Gamma \vdash (\nu\tilde{a} : \tilde{\mathsf{t}}\,; \Phi)P : (\nu\tilde{a} : \tilde{\mathsf{t}})\mathsf{T}} \qquad (\text{T-Par}) \quad \frac{\Gamma \vdash P : \mathsf{T} \quad \Gamma \vdash Q : \mathsf{S}}{\Gamma \vdash P|Q : \mathsf{T}|\mathsf{S}}$$

$\tilde{a}$ fulfills property Φ ($\mathsf{T}\downarrow_{\tilde{a}} \models \Phi$; in practical cases Φ is a shallow logic formula and this is actually spatial model checking). Note that, thanks to $\downarrow_{\tilde{a}}$, this is checked without looking at the environment: only the part of T that depends on $\tilde{a}$, that is $\mathsf{T}\downarrow_{\tilde{a}}$, is considered, the rest is masked. In particular, in $\mathsf{T}\downarrow_{\tilde{a}}$, any masked subterm that appears in parallel to the non-masked subterm can be safely discarded (a consequence of condition 1 of Ok). In this sense the type system is "local".

Rules for input and output are asymmetric, in the sense that, when typing a receiver $a(\tilde{x}).P$, the type information on P that depends on the input parameters $\tilde{x}$ is moved to the sender process. The reason is that the transmitted names $\tilde{b}$ are statically known only by the sender (rule (T-Out)). Accordingly, on the receiver's side (rule (T-Inp)), one only keeps track of the part of the continuation type that does not depend on the input parameters, that is T'. More precisely, the type of the continuation P is required to decompose – modulo type congruence – as $\mathsf{T}|\mathsf{T}'$, where T is the type prescribed by the context for a and T', which should not mention the input parameters $\tilde{x}$, is anything else. In essence, in well typed processes, all receivers on a must share a common part that deals with the received names $\tilde{x}$ as prescribed by the type T.

Finally, (T-Eq) is related to sub-typing. As mentioned in the Introduction, a key point of our system is that types should reflect the (shallow) spatial structure of processes. When considering sub-typing, this fact somehow forces us to abandon preorders in favor of an equivalence relation that respects P-sets membership, which leads to structural congruence. Further discussion on this point is found in Section 7.

The judgements derivable in this type system are written as $\Gamma \vdash_{\mathrm{L}} P : \mathsf{T}$.

Example 4. Consider the formula $\phi = \Box^*\neg \mathrm{H}^*(\overline{a}|\overline{b})$ saying that it is not possible to reach a configuration where both an output barb on a and one on b are available at the same time. Ok(ϕ) holds by Lemma 3. Consider the process $P = (\nu a, b : \mathsf{t}, \mathsf{t}; \phi)Q$, where: $\mathsf{t} = ()\mathbf{0}$, $Q = ((\overline{d}\langle a\rangle + \overline{d}\langle b\rangle)\,|\,!a.\overline{b}\,|\,!b.\overline{a})|d(x).\overline{x}$ and a context Γ s.t. $\Gamma \vdash d : (x : \mathsf{t})\overline{x} = \mathsf{t}'$. By applying the typing rules for input, output, summation and parallel composition:

$$\Gamma, a : \mathsf{t}, b : \mathsf{t} \vdash_{\mathrm{L}} Q \;\; : \;\; (\overline{d}.\overline{a} + \overline{d}.\overline{b})\,|\,!a(\mathsf{t}).\overline{b}\,|\,!b(\mathsf{t}).\overline{a}\,|\,d(\mathsf{t}') \;\triangleq\; \mathsf{T}.$$

$\mathsf{T}\downarrow_{a,b} = (\tau.\overline{a} + \tau.\overline{b})\,|\,!a(\mathsf{t}).\overline{b}\,|\,!b(\mathsf{t}).\overline{a}\,|\,\tau \models \phi$; hence, by (T-Res), $\Gamma \vdash_{\mathrm{L}} P : (\nu a, b : \mathsf{t}, \mathsf{t})\mathsf{T}$.

Basic properties. We state here the basic properties of the type system presented in the preceding subsection. Let us write $\Gamma \vdash_{\mathrm{NL}} P : \mathsf{T}$ if there exists a *normal* derivation of $\Gamma \vdash_{\mathrm{L}} P : \mathsf{T}$, that is, a derivation where the rule (T-Eq) is used only above rule (T-Inp). Modulo $\equiv$, every judgment derivable in the type system admits a normal derivation.

Proposition 2 (normal derivation). *If* $\Gamma \vdash_{\mathrm{L}} P : \mathsf{T}$ *then* $\Gamma \vdash_{\mathrm{NL}} P : \mathsf{S}$ *for some* $\mathsf{S} \equiv \mathsf{T}$.

Normal derivations are syntax-driven, that is, processes and their types share the same shallow structure. This fact carries over to all derivations, modulo $\equiv$. E.g., if $\Gamma \vdash_{\mathrm{L}} P : \mathsf{T}, \mathsf{T} \equiv (\tilde{\nu}\tilde{a} : \tilde{\mathsf{t}})(\mathsf{T}_1|\mathsf{T}_2)$ then $P \equiv (\tilde{\nu}\tilde{a} : \tilde{\mathsf{t}}; \tilde{\Phi})(P_1|P_2)$, with $\Gamma, \tilde{a} : \tilde{\mathsf{t}} \vdash_{\mathrm{L}} P_i : \mathsf{T}_i$, $i = 1, 2$.

Theorem 1 (subject reduction). $\Gamma \vdash_{\mathrm{L}} P : \mathsf{T}$ *and* $P \xrightarrow{\lambda} P'$ *implies that there exists a* T' *such that* $\mathsf{T} \xrightarrow{\lambda} \mathsf{T}'$ *and* $\Gamma \vdash_{\mathrm{L}} P' : \mathsf{T}'$.

5 Type Soundness for the Local System

In this section we prove a general type soundness result for our system and provide a few interesting examples of application of the type systems.

Definitions and results. We identify the general class of properties for which, at least in principle, model checking on processes can be reduced to a type checking problem whose solution requires only a (local) use of model checking on types. We do so by the following coinductive definition.

Definition 7 (locally checkable properties and formulae). *We let* Lc *be the largest predicate on P-sets such that whenever* $\mathrm{Lc}(\Phi)$ *then* $\mathrm{Ok}(\Phi)$ *and: (1) whenever* $\Gamma \vdash_{\mathrm{L}} P : \mathsf{T}$ *and* $\tilde{x} \supseteq \mathrm{supp}(\Phi)$ *and* $\mathsf{T} \downarrow_{\tilde{x}} \models \Phi$ *then* $P \models \Phi$*; (2)* $\mathrm{Lc}(\Phi_\lambda)$ *holds for each* λ*. If* $\mathrm{Lc}(\Phi)$ *then we say* Φ *is* locally checkable.

A formula $\phi \in \mathcal{F}$ is said to be locally checkable if $[\![\phi]\!]$ is locally checkable.

Theorem 2 (run-time soundness). *Suppose that* $\Gamma \vdash_{\mathrm{L}} P : \mathsf{T}$ *and that P is decorated with locally checkable P-sets only. Then* $P \to^* P'$ *implies that* P' *is well-annotated.*

Our task is now providing sufficient syntactic conditions on formula ϕ that guarantee $\mathrm{Lc}([\![\phi]\!])$.

Lemma 4. *Suppose* $\Gamma \vdash_{\mathrm{L}} P : \mathsf{T}$*. (a) If* $\phi \in \mathcal{F}^-_{\tilde{x}}$ *and* $\mathsf{T} \downarrow_{\tilde{x}} \models \phi$ *then* $P \models \phi$*. (b) If* $\phi \in \mathcal{F}^+_{\tilde{x}}$ *and* $P \models \phi$ *then* $\mathsf{T} \downarrow_{\tilde{x}} \models \phi$*.*

Theorem 3. *Any negative formula of the form* $\Box^*\phi$ *is locally checkable.*

The above result automatically provides us a type soundness result for an interesting class of formulae, that include both safety and liveness properties.

Examples. The formulae *NoRace*(a) and *UniRec*(a) fits in the format given by Theorem 3, hence they are locally checkable. As an example, consider

$$P = (\nu a,b,c : ()\mathbf{0}, \mathsf{t}', \mathsf{t}\,;\, \mathit{UniRec}(a))((\overline{c}\langle a\rangle \mid a + b(x).x) \mid c(y).\overline{b}\langle y\rangle)$$

where $\mathsf{t} = (x)\overline{b}.x$ and $\mathsf{t}' = (y)y$. By the typing rules, we easily derive

$$\Gamma, a : ()\mathbf{0}, b : \mathsf{t}', c : \mathsf{t} \vdash_{\mathrm{L}} ((\overline{c}\langle a\rangle \mid a + b(x).x) \mid c(y).\overline{b}\langle y\rangle) : \mathsf{T}$$

with $\mathsf{T} \triangleq \overline{c}.\overline{b}.a \mid a + b(\mathsf{t}') \mid c(\mathsf{t})$. Since $\mathsf{T} \downarrow_{a,b,c} = \mathsf{T} \models \mathit{UniRec}(a)$, we can apply (T-Res) and get

$$\Gamma \vdash_{\mathrm{L}} P : (\nu a,b,c : ()\mathbf{0}, \mathsf{t}', \mathsf{t})\mathsf{T}.$$

For another example, consider the following access policy for a shared resource c. Before using the resource, a lock l must be acquired; the resource must then be used immediately, and the lock must be released immediately after that. If we identify an available resource c with an input barb on c, a use of c with a synchronization on c and the availability of l with an output barb on l, the above policy can be described by the following

formula, where $[c]$ stands for $\neg\langle c\rangle\neg$: $SafeLock(l,c) \triangleq \Box^*((\overline{l} \to c) \ \wedge \ [c]\overline{l})$. This is a negative formula fitting the format of Theorem 3, hence it is locally checkable. As an example of use of this formula, the process $Q = (\nu c, l; SafeLock(l,c))(\overline{l}|c|\overline{a}\langle l,c\rangle) \,|\, a(x,y).!x.(\overline{y}.y|\overline{x})$ is well typed under a Γ s.t. $\Gamma \vdash a : (x,y)!x.(\overline{y}.y|\overline{x})$. Note that neither (the analog of) $UniRec(a)$, nor $SafeLock(l,c)$ is included in the type soundness theorem of [10].

Finally, note that $Resp(a)$ and $DeadFree(a)$ do not fit the format of Theorem 3. Indeed, these formulae are not locally checkable. E.g., consider $R = (\nu a; Resp(a))(c.a|\overline{a})$. This process is easily seen to be well-typed under $c : ()\mathbf{0}$, simply because the c blocking a is masked (turned into τ) in (T-Res). However, $c.a|\overline{a}$ clearly fails to satisfy $Resp(a)$.

6 A "Global" Type System

The $Resp(a)$ example at the end of the preceding section makes it clear that it is not possible to achieve type soundness result for properties like responsiveness unless we drop the "locality" condition in the restriction rule. Indeed, those properties can only be checked if one can look at the part of the type involving names from which the restricted ones causally depend. In the previous example, where $\mathsf{T} = c.a|\overline{a}$, this means checking $Resp(a)$ against $\mathsf{T} \downarrow_{a,c} = \mathsf{T}$, rather than against $\mathsf{T} \downarrow_a$, thus detecting the failure of the property.

Below, we introduce a new type system that pursues this idea. Note that dropping locality implies some loss of compositionality and effectiveness. The type system relies on the use of dependency graphs, a technical device, introduced in the next subsection, which helps to individuate causal relations among names.

Dependency graphs. Let χ range over a set $\mathfrak{a} = \{\epsilon, \circ, \bullet\}$ of *annotations*. For $I \subseteq \mathcal{N}$, we let a set of annotated names $\hat{I}$ be a total function from I to $\mathfrak{a}$; by slight abuse of notation, we write $a^\chi \in \hat{I}$ rather than $\hat{I}(a) = \chi$. The informal meaning of annotations is: ϵ = free name, $\circ$ = input-bound name, $\bullet$ = restricted name. A *dependency graph* G is a pair $\langle V, E\rangle$, where: $V = \hat{I} \cup W$, with $W \subseteq \{(\nu\tilde{x}) \,|\, \tilde{x} \subseteq \mathcal{N}\}$, is a set of annotated names and restrictions representing *vertices*, and $E \subseteq V \times V$ is a set of *edges*.

A dependency graph $G = \langle V, E\rangle$, with $V = \hat{I} \cup W$ ranged over by $u, v, \ldots$, encodes causal relations among (free or bound) names in I. Vertices of the form $(\nu\tilde{x})$ are introduced for delimiting the scope of restrictions. Edges $(u,v) \in E$ are also written as $u \to_{\mathrm{G}} v$; $\to_{\mathrm{G}}^*$ is the reflexive and transitive closure of $\to_{\mathrm{G}}$. A *root* of G is a vertex $u \in V$ such that for no v, $v \to_{\mathrm{G}} u$; the set of G's roots is denoted by $\mathrm{roots}(G)$. Given a dependency graph $G = \langle V, E\rangle$, with $V = \hat{I} \cup W$, a name a is *critical in G with respect to $\tilde{b}$*, if it belongs to the set of names $G(\tilde{b})$ defined below.

$$G(\tilde{b}) \triangleq \{x \,|\, x^\epsilon \in \hat{I} \wedge \exists x \to_{\mathrm{G}} v_1 \to_{\mathrm{G}} \cdots \to_{\mathrm{G}} v_n = b \in \tilde{b} \text{ s.t. } \forall 1 \le i < n : \ v_i = (\nu\tilde{y}) \text{ implies } b \notin \tilde{y}\}.$$

The set of *critical names* in G, written $\mathrm{cr}(G)$, is defined as $\mathrm{cr}(G) \triangleq \bigcup_{b^\bullet \in \hat{I}} G(b)$. Finally, we define $G[\tilde{b}] \triangleq G(\tilde{b}) \cup \tilde{b}$.

In order to define dependency graphs associated to types, we introduce three auxiliary operations on graphs: (i) union $G_1 \cup G_2$ is defined componentwise as expected, provided the sets of vertices V_1 and V_2 agree on annotations (otherwise union is not defined); (ii)

χ-update $G\uparrow^{\chi}_{\tilde{x}}$ changes into χ the annotation of all names in $\tilde{x}$ occurring in V; (iii) a-rooting is defined as $a \to G \triangleq \langle\, V \cup \{a^{\epsilon}\}, E \cup \{(a,b) \mid b \in \mathrm{roots}(G)\}\,\rangle$, where $G = \langle V, E\rangle$, provided a does not occur in V with annotations different from ϵ (otherwise a-rooting is not defined); (iv) $(\nu\tilde{x})$-rooting is defined as $(\nu\tilde{x}) \to G \triangleq \langle\, V\,, E \cup \{((\nu\tilde{x}), b) \mid b \in \mathrm{roots}(G)\}\,\rangle$. Dependency graphs are inductively defined over types in *normal form*. Let us say a type is *prime* if it is of the form either $\sum_{i\in I}\mu_i.\mathsf{T}_i$ with $I \neq \emptyset$ or $!a(\mathsf{t}).\mathsf{T}$. Let us say a type is in *head normal form* if it is of the form $(\tilde{\nu a})(\mathsf{T}_1 | \cdots | \mathsf{T}_n)$ with the T_i's prime and in *normal form* if the T_i's are recursively in normal form. Similar definition for processes. For any T and t in normal form, the dependency graphs G_{T} and G_{t} are defined by mutual induction on the structure of T and t as follows (it is assumed that in T and t bound names are distinct from each other and from free names).

$$G_{\overline{a}.\mathsf{T}} = a \to G_{\mathsf{T}} \qquad G_{a(\mathsf{t}).\mathsf{T}} = a \to (G_{\mathsf{t}} \cup G_{\mathsf{T}}) \qquad G_{!a(\mathsf{t}).\mathsf{T}} = G_{a(\mathsf{t}).\mathsf{T}}$$

$$G_{\sum_{i\in I}\mu_i.\mathsf{T}_i} = \bigcup_{i\in I} G_{\mu_i.\mathsf{T}_i} \quad |I| \neq 1 \qquad G_{\prod_i \mathsf{T}_i} = \bigcup_i G_{\mathsf{T}_i}$$

$$G_{(\nu\tilde{x}:\tilde{\mathsf{t}})\mathsf{T}} = (\nu\tilde{x}) \to ((G_{\mathsf{T}} \cup \bigcup_{\mathsf{t}\in\tilde{\mathsf{t}}} G_{\mathsf{t}})\uparrow^{\bullet}_{\tilde{x}}) \qquad G_{(\tilde{x}:\tilde{\mathsf{t}})\mathsf{T}} = (G_{\mathsf{T}} \cup \bigcup_{\mathsf{t}\in\tilde{\mathsf{t}}} G_{\mathsf{t}})\uparrow^{\circ}_{\tilde{x}}\ .$$

In essence, G_{T} encodes potential causal dependencies among (free or bound) names of T as determined by prefixes in T. In the sequel, we shall abbreviate $\mathrm{cr}(G_{\mathsf{T}})$ and $G_{\mathsf{T}}[\tilde{b}]$, for some $\tilde{b} \subseteq \mathrm{fn}(\mathsf{T})$, as $\mathrm{cr}(\mathsf{T})$ and $\mathsf{T}[\tilde{b}]$, respectively.

Typing rules. We need some additional notations. A channel type $(\tilde{x})\mathsf{T}$ is said to be *well-formed* if $\tilde{x}\#\mathrm{cr}(\mathsf{T})$; in what follows, we only consider contexts Γ containing well-formed channel types. For any type T we let $\mathsf{T}\Downarrow_{\tilde{x}}$ denote $\mathsf{T}\downarrow_{\mathsf{T}[\tilde{x}]}$ (note that $\mathrm{fn}(\mathsf{T}\Downarrow_{\tilde{x}}) = \mathsf{T}[\tilde{x}]$ by definition). Intuitively, in $\mathsf{T}\Downarrow_{\tilde{x}}$, we keep the names in $\tilde{x}$ and those that are causes of $\tilde{x}$ in T; the others are masked. We also need a more permissive notion of well-annotated process, that allows re-arranging of top-level restrictions before checking annotations (property). To see why this is necessary, consider $\phi = \Box^*(\Diamond^* a | \Diamond^* a)$, a typical property one would like to check in the new system. Consider the processes $P = (\nu b)(\nu a;\phi)R$ and $Q = (\nu a;\phi)(\nu b)R$, with $R = b.\overline{c} \mid b.\overline{d} \mid \overline{b} \mid \overline{b} \mid c.a \mid d.a$. We observe that $(\nu b)R \not\models \phi$, so that Q is not well-annotated according to Definition 5; on the other hand, $Q \equiv P$ and $R \models \phi$, which suggests that P, hence Q, could be considered as well-annotated up to a swapping of (νa) and (νb).

Definition 8 (globally well-annotated processes). *A process $P \in \mathcal{P}$ is* globally well-annotated *if whenever $P \equiv (\tilde{\nu b})(\nu\tilde{a};\, \Phi)(\tilde{\nu c})Q$, with Q a parallel composition of prime processes, then there is a permutation $\tilde{b}'\,\tilde{c}'$ of $\tilde{b}\,\tilde{c}$ such that $P \equiv (\tilde{\nu b'})(\nu\tilde{a};\, \Phi)(\tilde{\nu c'})Q$ and $(\tilde{\nu c'})Q \models \Phi$.*

The global type system is obtained by replacing some rules of the local one (Table 4) with those reported in Table 5. The type system makes use of an auxiliary relation $\propto_{\tilde{x}}$ among P-sets and types, defined coinductively as follows (the use of this relation is explained in the sequel).

Definition 9 ($\propto_{\tilde{x}}$). *We let $\propto_{\tilde{x}}$ be the largest relation on P-sets and types such that whenever $\Phi \propto_{\tilde{x}} \mathsf{T}$ then: (1) $\mathsf{T}\downarrow_{\mathsf{T}[\tilde{x}]} \models \Phi$; (2) for each λ, T' such that $\mathsf{T}\downarrow_{\mathsf{T}[\tilde{x}]} \xrightarrow{\lambda} \mathsf{T}'\downarrow_{\mathsf{T}[\tilde{x}]}$ then $\Phi_\lambda \propto_{\tilde{x}} \mathsf{T}'$.*

Table 5. Typing rules

$$\text{(T-Res)}\ \frac{\Gamma, \tilde{a} : \tilde{\mathsf{t}} \vdash P : \mathsf{T} \quad \Phi \propto_{\tilde{a}} \mathsf{T}}{\Gamma \vdash (\nu \tilde{a} : \tilde{\mathsf{t}}; \Phi) P : (\nu \tilde{a} : \tilde{\mathsf{t}})\mathsf{T}} \qquad \text{(T-Eq-P)}\ \frac{\Gamma \vdash P : \mathsf{T} \quad P \equiv Q}{\Gamma \vdash Q : \mathsf{T}}$$

$$\text{(T-Rep)}\ \frac{\Gamma \vdash P : \mathsf{T} \quad \mathrm{cr}(\mathsf{T}) = \emptyset}{\Gamma \vdash !P : !\mathsf{T}} \qquad \text{(T-Par)}\ \frac{\Gamma \vdash P : \mathsf{T} \quad \Gamma \vdash Q : \mathsf{S} \quad \mathrm{cr}(\mathsf{T}) \# \mathsf{S} \quad \mathrm{cr}(\mathsf{S}) \# \mathsf{T}}{\Gamma \vdash P | Q : \mathsf{T} | \mathsf{S}}$$

$$\text{(T-Out)}\ \frac{\Gamma \vdash a : (\tilde{x} : \tilde{\mathsf{t}})\mathsf{T} \quad \Gamma \vdash \tilde{b} : \tilde{\mathsf{t}} \quad \Gamma \vdash P : \mathsf{S} \quad \tilde{b} \# \mathrm{cr}(\mathsf{T}) \quad \mathrm{cr}(\mathsf{T}[\tilde{b}/\tilde{x}]) \# \mathsf{S} \quad \mathsf{T}[\tilde{b}/\tilde{x}] \# \mathrm{cr}(\mathsf{S})}{\Gamma \vdash \overline{a}\langle \tilde{b} \rangle.P : \overline{a}.(\mathsf{T}[\tilde{b}/\tilde{x}] \,|\, \mathsf{S})}$$

Note the presence of a new structural rule for processes, (T-Eq-P) forcing subject congruence, which is not derivable from the other rules of the system. As an example, while $P = (\nu a : \mathsf{t}; Resp(a))(b.a|\overline{b}|\overline{a})$ can be typed without using rule (T-Eq-P), the structurally congruent process $(\nu a : \mathsf{t}; Resp(a))(b.a|\overline{a})|\overline{b}$ *could not* be typed without using that rule. The condition on critical names in rule (T-Par) ensures that any Q put in parallel to P will not break well-annotated-ness of P (and vice-versa). A similar remark applies to the rules for output and replication. In rule (T-Res), use of the relation $\propto_{\tilde{a}}$ ensures that each derivative of T satisfies the corresponding derivative of Φ. It is worth noticing that checking $\Phi \propto_{\tilde{a}} \mathsf{T}$ could be undecidable, given that in general we are in the presence of infinite state systems: at the end of the next section, we will identify a class of formulas for which checking $[[\phi]] \propto_{\tilde{a}} \mathsf{T}$ reduces to checking $\mathsf{T} \Downarrow_{\tilde{a}} \models \phi$. The judgements derivable in the new type system are written as $\Gamma \vdash_{\mathrm{G}} P : \mathsf{T}$. It is worth noticing that the system introduced in Table 5 is not syntax-driven, but a syntax-directed version can be easily defined by adding some constraints on the structure of processes in the premises of the typing rule for parallel composition (we omit the details for lack of space).

Type Soundness. Similarly to the local case, we identify a general class of properties for which, at least in principle, model checking on processes can be reduced to a type checking problem whose solution requires only model checking on types, then give sufficient syntactic conditions for global-checkable-ness. The definition of *globally checkable property* (omitted) is the same as the local one, except that the local hiding operator "$\downarrow_{\tilde{x}}$" is replaced by "$\Downarrow_{\tilde{x}}$".

Theorem 4 (run-time soundness). *Suppose that $\Gamma \vdash_{\mathrm{G}} P : \mathsf{T}$ and that P is decorated with globally checkable P-sets only. Then $P \rightarrow^* P'$ implies that P' is globally well-annotated.*

Like in the local case, we can give syntactic conditions for a formula to be globally checkable.

Theorem 5. *Suppose ϕ is of the form: (a) $\Box^* \psi$ with negation not occurring underneath any $\langle -\tilde{y} \rangle^*$ in ψ; or (b) $\Box^*_{-\tilde{y}} \Diamond^* \psi'$, with negation not occurring in ψ'. Then ϕ is globally checkable.*

The following proposition guarantees that for formulas that satisfy the premises of Theorem 5 checking $[[\phi]] \propto_{\tilde{a}} \mathsf{T}$ reduces to checking $\mathsf{T} \Downarrow_{\tilde{a}} \models \phi$.

Proposition 3. *Suppose $\phi \in \mathcal{F}_{\tilde{x}}$ is of the form of the form (a) and (b) as specified in Theorem 5. If $\mathsf{T} \Downarrow_{\tilde{x}} \models \phi$ then $[[\phi]] \propto_{\tilde{x}} \mathsf{T}$.*

Examples. All properties defined in Example 1 fit the format of Theorem 5, hence they are globally checkable. As an example, consider $P = (\nu a : Resp(a))(\overline{c}\langle a\rangle)|Q$, where $Q = !c(x).(\overline{x}|x)|\overline{c}\langle b\rangle$. Under a suitable Γ, we derive $\Gamma \vdash_{\mathrm{G}} \overline{c}\langle a\rangle|Q : \overline{c}.(\overline{a}|a)|!c|\overline{c}.(\overline{b}|b) \triangleq \mathsf{T}$. Since $\mathsf{T} \downarrow_{\mathsf{T}[a]} = \overline{c}.(\overline{a}|a)|!c|\overline{c}.(\tau|\tau) \models Resp(a)$, by (T-Res), we get $\Gamma \vdash_{\mathrm{G}} (\nu a : Resp(a))(\overline{c}\langle a\rangle|Q) : (\nu a)\mathsf{T}$, hence we can conclude that $\Gamma \vdash_{\mathrm{G}} P : (\nu a)\mathsf{T}$ using (T-Eq-P).

It is worth to notice that (the analogs of) responsiveness and deadlock freedom escape the type soundness theorem of [10], although, for deadlock freedom, a soundness result can still be proven by ad-hoc reasoning on certain basic properties of the system.

7 Discussion

We discuss here some limits, and possible workarounds, of our approach, and contrast them with the generic type system approach of [10]. In [10], the subtyping relation makes an essential use of a "sub-divide" law, $\mathsf{T} \equiv \mathsf{T} \uparrow_{\tilde{x}} | \mathsf{T} \downarrow_{\tilde{x}}$. This rule allows one to split *any* type into a part depending only on $\tilde{x}$, $\mathsf{T} \downarrow_{\tilde{x}}$, and a part not depending on $\tilde{x}$, $\mathsf{T} \uparrow_{\tilde{x}}$. As an example, with this law one has $a.b.\overline{x} \equiv a.b.\tau|\tau.\tau.\overline{x}$. This law enhances the flexibility of the input rule, hence of the type system. On the other hand, it disregards the spatial properties of terms, leading to a lack of structural correspondence between types and processes. In our system, we stick to spatial-preserving laws, thus trading off some flexibility for precision. As seen, this gain in precision has influential consequences on the class of properties for which type soundness can be proven (e.g., the class includes interesting liveness properties). An example of process that cannot be treated in our type systems because of the absence of the "sub-divide" law is the process $Q = !a(x).(\nu c)\Big(b(y).((\nu z)(\overline{c}\langle x,z\rangle|z.\overline{y})|c(x,z).(\overline{x}|\overline{z}))\Big)$. Here, a can be viewed as an invocation channel, x as a formal invocation parameter and y as an acknowledgement channel, introduced by another input (on b). It appears that y and x are related (via c), which makes the type of b dependent on the bound name x, which cannot be expressed in our system. This dependency could be discarder using the sub-divide law. In the example, the very dependency of y from x suggests a way to re-write the process into a conceptually equivalent one that can be dealt with in our systems. E.g., $!a(x,y).(\nu c)\Big((\nu z)(\overline{c}\langle x,z\rangle|z.\overline{y})|c(x,z).(\overline{x}|\overline{z})\Big)$.

8 Conclusion, Further and Related Work

We have provided a framework that incorporates ideas from both spatial logics and behavioural type systems, drawing benefits from both. Implementation issues are not in the focus of this paper. In this respect, the normal derivation property already provides us with syntax driven systems. Of course, implementing the model checks $\mathsf{T} \models \phi$ is an issue. One possibility would be re-using existing work on spatial model checking: Caires' work [6] seems to be a promising starting point. Also, approximations of possibly infinite-state ccs types with Petri Nets, or even finite-state automata, in the

vein of [12], seem unavoidable to obtain effective tools. Finally, it would be interesting to cast our approach in more applicative scenarios, like calculi for service-oriented computing [1].

Apart from the already cited works, also related to our approach are some recent proposals by Caires. In [4, 5], a logical semantics approach to types for concurrency is pursued. Closest to our work is [4], where a generic type system for the pi-calculus - parameterized on the subtyping relation - is proposed. The author identifies a family of types, the so called shared types, which allow to modularly and safely compose spatial and shared (classical invariants) properties and to safely factorize spatial properties. A preliminary investigation of the ideas presented in this paper, in a much simpler setting, is in [2].

References

1. Acciai, L., Boreale, M.: A type system for client progress in a service-oriented calculus. In: Degano, P., et al. (eds.) Montanari Festschrift. LNCS, vol. 5065, pp. 642–658. Springer, Heidelberg (2008)
2. Acciai, L., Boreale, M.: Type abstractions of name-passing processes. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 302–317. Springer, Heidelberg (2007)
3. Acciai, L., Boreale, M.: Responsiveness in process calculi. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 136–150. Springer, Heidelberg (2008)
4. Caires, L.: Logical Semantics of Types for Concurrency. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 16–35. Springer, Heidelberg (2007)
5. Caires, L.: Spatial-Behavioral Types, Distributed Services, and Resources. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 98–115. Springer, Heidelberg (2007)
6. Caires, L.: Behavioral and Spatial Observations in a Logic for the pi-Calculus. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 72–89. Springer, Heidelberg (2004)
7. Caires, L., Cardelli, L.: A spatial logic for concurrency (part I). Inf. Comput. 186(2), 194–235 (2003)
8. Chaki, S., Rajamani, S.K., Rehof, J.: Types as models: model checking message-passing programs. In: POPL 2002, pp. 45–57 (2002)
9. Cardelli, L., Gordon, A.D.: Anytime, Anywhere: Modal Logics for Mobile Ambients. In: POPL 2000, pp. 365–377 (2000)
10. Igarashi, A., Kobayashi, N.: A generic type system for the Pi-calculus. Theor. Comput. Sci. 311(1-3), 121–163 (2004)
11. Kobayashi, N.: Type-based information flow analysis for the pi-calculus. Acta Inf. 42(4-5), 291–347 (2005)
12. Kobayashi, N., Suenaga, K., Wischik, L.: Resource Usage Analysis for the pi-Calculus. Logical Methods in Computer Science 2(3) (2006)
13. Kobayashi, N., Suto, T.: Undecidability of 2-Label BPP Equivalences and Behavioral Type Systems for the pi -Calculus. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 740–751. Springer, Heidelberg (2007)
14. Milner, R.: The polyadic π-calculus: a tutorial. In: Logic and Algebra of Specification, pp. 203–246. Springer, Heidelberg (1993)
15. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980)
16. Sangiorgi, D.: The Name Discipline of Uniform Receptiveness. Theoretical Computer Science 221(1-2), 457–493 (1999)

A Spatial Equational Logic for the Applied π-Calculus

Étienne Lozes and Jules Villard

LSV, ENS Cachan, CNRS 61 av. du pdt Wilson, 94230 Cachan, France
{lozes,villard}@lsv.ens-cachan.fr

Abstract. Spatial logics have been proposed to reason locally and modularly on algebraic models of distributed systems. In this paper we define the spatial equational logic AπLwhose models are processes of the applied π-calculus. This extension of the π-calculus allows term manipulation and records communications as active substitutions in a frame, thus augmenting the underlying predefined equational theory. Our logic allows one to reason locally either on frames or on processes, thanks to static and dynamic spatial operators. We study the logical equivalences induced by various relevant fragments of AπL, and show in particular that the whole logic induces a coarser equivalence than structural congruence. We give characteristic formulae for some of these equivalences and for static equivalence. Going further into the exploration of AπL's expressivity, we also show that it can eliminate standard term quantification.

1 Introduction

Spatial logics. Spatial logics, partly inspired by pioneering ideas of resource logics [1,2], have been proposed to reason locally and modularly on algebraic models of distributed systems such as ambients [3] or π-calculus [4]. Two essential connectives in these logics are spatial conjunction and adjunct. The spatial conjunction $A * B$, which introduces local reasoning, is the cause of a very intensional discriminating power for spatial logics [5,6], as usually logical equivalence is structural congruence. When this connective is dropped, reasoning only with adjunct $-\!\!*$ yields extensional equivalences such as barb equivalence [7]. Though quite intuitive, these results are dependent on the nature of the process model they deal with, and should be treated with some care: the intensional equivalence might be much coarser than structural congruence, and logical equivalence does not have to be a congruence in the presence of adjunct.

A spatial equational logic. In this paper we investigate a spatial equational logic that extends the first order equational logic with spatial connectives. Term equality $M = N$ is defined by both global axioms from an equational theory $\mathcal{E}$ and local axioms from a frame. For example, the frame $\Phi = \{^{\mathrm{enc}(s,y)}/_{x}\} \,|\, \{^{\mathrm{pk}(n)}/_{y}\}$, that we will use as an example throughout this paper, augments the equational theory with the knowledge that x is an alias for a message encrypted with a public key, itself aliased by y. To reflect the private nature of s and n, these names will be hidden, and we will write $\nu n, s.\, \Phi$. Spatial conjunction $*$ may then split the frame into smaller pieces that may not share any secret. For instance, $\nu n, s.\, \Phi = (\nu s.\, \{^{\mathrm{enc}(s,y)}/_{x}\}) * (\nu n.\, \{^{\mathrm{pk}(n)}/_{y}\})$.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 387–401, 2008.

Our spatial equational logic, written AπL, naturally arises as the spatial logic of the applied π-calculus [8], an extension of π-calculus [9] where processes may communicate terms through channels. One peculiar aspect of applied π-calculus is that what is sent to the environment is kept as *active substitutions* that act as local aliases that extend the equational theory. For instance, $\nu n, s.\, \Phi$ might have been generated from a process $\nu n.\, \{^{\mathrm{pk}(n)}/_{y}\} \mid \nu s.\, \bar{a}\langle \mathrm{enc}(s, y)\rangle.P$ sending $\mathrm{enc}(s, y)$ to an environment listening on channel a, thus reducing to $\nu n, s.\, \Phi \mid P$.

Motivations. Cryptographic protocols are the most standard applications modeled with the applied π-calculus. Observation by a passive attacker may be modeled by a static observational equivalence, usually simply called static equivalence. This proved sound enough to express some complex cryptographic properties such as resistance to guessing attacks, through other observational equivalences related to static equivalence. Logical foundations for these notions of observations are however needed to understand their dependencies and provide more flexible specifications.

From a logical point of view, spatial logics can be considered as fragments of second order logics, hence have to be compared to first-order and second-order logics regarding expressiveness issues. This line of research has been actively followed [10,11,12], but is still open regarding first order equational logics.

Contributions. Our first contribution is the characterization of the logical equivalences of the static, static extensional, and dynamic fragments. Static fragments turn out to play similar roles as in the case of the π-calculus: static intensional equivalence is proved to coincide with structural congruence for frames, whereas static extensional logical equivalence coincides with static equivalence. This is, to our knowledge, the first logical characterization of static equivalence that is independent on the equational theory. All the constructions involved are rather simple compared to other logical characterizations of frame equivalence [13] for specific classes of equational theories. Moreover, characteristic formulae are derivable for both intensional and extensional equivalences.

Our second contribution deals with the logical equivalence for the dynamic intensional fragment. Surprisingly, we show that the logic cannot distinguish messages with similar information content. As a consequence, this equivalence is coarser than mere structural congruence. Moreover, we show that it is not a congruence, due to the possible introduction of noise in communications that the logic may not detect. This noticeably complicates the techniques to obtain an axiomatization of this equivalence. We point out some admissible axioms for logical equivalence and prove this axiomatization complete for the equational theory of finite trees.

Our third contribution is a quantifier elimination technique that shows that standard term quantifiers $\exists t.\, A$ can be mimicked by spatial connectives, which illustrates one more particular example of a spatial logic that is more expressive than first-order logic.

Structure of the paper. In Section 2, we collect all the necessary background on the applied π-calculus, and define our process compositions $*$ and $\natural$. Section 3 introduces AπL; Sections 4 and 5 present the characterizations of the logical equivalences for the static and dynamic fragments respectively. Section 6 establishes the quantifier elimination property.

2 Applied π-Calculus

2.1 Terms

The grammar of applied π-calculus processes relies on the definition of a set of *terms* along with an *equational theory*. This lets the user decide which cryptographic primitives the calculus will use for example. The set of terms is constructed using disjoint infinite sets $\mathcal{V}$ and $\mathcal{N}$ of respectively variables and names, and a finite *signature* Σ, which is a set of functions, each with its arity (constants have arity 0). Its grammar is as follows, $ar(f)$ being the arity of f:

$$M, N ::= x \in \mathcal{V} \mid a \in \mathcal{N} \mid f(M_1, \ldots, M_{ar(f)}) .$$

We will use the letters a, b, c, n, m, s to refer to elements of $\mathcal{N}$, x, y, z for elements of $\mathcal{V}$ and u, v, w for "meta-variables" that may belong either to $\mathcal{N}$ or $\mathcal{V}$. We will write M, N for terms.

These terms are equipped with an equivalence relation $\mathcal{E}$ called an equational theory on Σ, where membership of a pair (M, N) of terms is written $\mathcal{E} \vdash M = N$, or simply $M = N$ if $\mathcal{E}$ is clear from context. This relation must be closed under substitution of terms for variables or names ($M_1 = M_2$ implies $M_1[u \leftarrow N] = M_2[u \leftarrow N]$) and context application ($N_1 = N_2$ implies $M[x \leftarrow N_1] = M[x \leftarrow N_2]$). $fn(M)$ and $fv(M)$ are respectively the sets of free names and free variables of M, defined as usual, and $fnv(M) \triangleq fn(M) \cup fv(M)$.

2.2 Processes

Applied π-calculus extends the standard π-calculus with primitives for term manipulation, namely *active substitutions* and term communications. The grammar of processes is split into two levels: the *plain* processes which account for the dynamic part, and the *extended* ones, also simply referred to as "processes" which extend the former with a *static* part.Note that replication $!P^p$ is not part of our setting.

$P^p, Q^p, \ldots ::=$	plain processes	$P, Q, \ldots ::=$	(extended) processes
$\mathbf{0}$	null process	P^p	plain process
$P^p \mid Q^p$	composition	$\{^M/_x\}$	active substitution
$\nu a.\, P^p$	name restriction	$P \mid Q$	parallel composition
$a^{ch}(n).P^p$	name input	$\nu a.\, P$	name restriction
$\bar{a}^{ch}\langle n \rangle.P^p$	name output	$\nu x.\, P$	variable restriction
$a(x).P^p$	term input		
$\bar{a}\langle M \rangle.P^p$	term output		
$if\ M = N$ $then\ P^p\ else\ Q^p$	conditional		

Our grammar differs from the original one in that it allows communications of two kinds, that may not interfere: communications of names behave as in the standard π-calculus whereas communications of terms may interact with active substitutions and

conditionals. Names are thus allowed to serve both as channels through which communications may occur and as atoms on which to build terms, without the need to rely on a type system to ensure the validity of communications.

From now on, we will only consider extended processes whose active substitutions are cycle-free. Furthermore, we will always assume that there is at most one active substitution for each variable, and *exactly one* if the variable is restricted. A process following these constraints will be called well-formed.

The set of free names (resp. variables) of a process P is defined as usual and written $fn(P)$ (resp. $fv(P)$), with $fn(\{^M/_x\}) \triangleq fn(M)$ (resp. $fv(\{^M/_x\}) \triangleq \{x\} \cup fv(M)$), and with both restrictions and both inputs being binders. We write $fnv(P)$ for the set $fn(P) \cup fv(P)$.

Compositions of active substitutions of the form $\{^{M_1}/_{x_1}\} \mid \cdots \mid \{^{M_n}/_{x_n}\}$ will be written $\{^{\boldsymbol{M}}/_{\boldsymbol{x}}\}$, and referred to using σ, τ. Depending on the context, we will write $\boldsymbol{x}$ for both the vector $x_1, \ldots, x_n$ and the associated set $\{x_1, \ldots, x_n\}$, where $n = |\boldsymbol{x}|$. Trailing $\mathbf{0}$'s in processes will often be omitted, as well as null $else$ branches in conditionals.

2.3 Operational Semantics

The structural congruence relation $\equiv$ identifies processes that can be obtained one from another by mere rewriting. It is the smallest equivalence relation on well-formed extended processes that is stable by α-conversion on both names and variables and by application of contexts, and that satisfies the usual rules w.r.t. the AC properties of $\mid$ with neutral $\mathbf{0}$, and the scope extrusion of restrictions. For instance, we have that $\nu a.\,(\nu x.\,\{^a/_x\} \mid \bar{a}\langle b\rangle) \mid \bar{c}\langle y\rangle \;\equiv\; \nu x.\,\nu a.\,(\{^a/_x\} \mid \bar{a}\langle b\rangle \mid \bar{c}\langle y\rangle)$. In addition, it must satisfy the following rules:

$$\begin{array}{ll} \textsc{Alias} & \nu x.\,(\{^M/_x\} \mid P) \equiv P[x{\leftarrow}M] \\ \textsc{Subst} & \{^M/_x\} \mid P^p \equiv \{^M/_x\} \mid P^p[x{\leftarrow}M] \\ \textsc{Rewrite} & \{^M/_x\} \equiv \{^N/_x\} \text{ if } \mathcal{E} \vdash M = N \end{array}$$

Here, a context (resp. an evaluation context) is an extended process with a hole in place of a plain (resp. extended) process. This hole can be filled with any extended process provided the resulting extended process is well-formed. The original structural congruence [8] is closed by application of evaluation contexts instead of arbitrary contexts. This makes inductive characterization of processes up to $\equiv$ impossible (see Section 5).

Due to the Rewrite rule, two structurally congruent processes may not have the same set of free names or variables. Thus, we define the closures $\overline{fn}(P), \overline{fv}(P), \overline{fnv}(P)$ of these sets up to structural congruence, and the corresponding sets for terms. For instance, $\overline{fn}(P) \triangleq \bigcap_{Q \equiv P} fn(Q)$ and $\overline{fv}(M) \triangleq \bigcap_{N=M} fv(N)$.

It is worth mentioning that rules Alias and Subst are not the ones from the original applied π-calculus, namely:

$$\begin{array}{ll} \textsc{Alias'} & \nu x.\,\{^M/_x\} \equiv \mathbf{0} \\ \textsc{Subst'} & \{^M/_x\} \mid P \equiv \{^M/_x\} \mid P[x{\leftarrow}M] \end{array}$$

With our rules, active substitutions may affect other active substitutions only if their domain is a restricted variable. They may only apply to plain processes otherwise. As

such, the rule ALIAS' is still valid in our setting, whereas SUBST' is restricted to plain processes only. This makes our quantifier elimination technique easier, as it dramatically limits the interferences between active substitutions. On the other hand, it does not change the behaviour of processes, as both the reduction rules and the static equivalence definitions, presented below, stay the same. Moreover, a process P can still always be rewritten into a set of restricted names $\boldsymbol{n}$, a public composition of active substitutions σ and a public plain process Q: $P \equiv \nu \boldsymbol{n}.\,(\sigma \mid Q)$.

Finally, internal reduction $\rightarrow$ is the smallest relation that satisfies the rules below and that is closed by structural congruence and by application of evaluation contexts.

$$\begin{array}{ll}
\textsc{Comm-T} & \bar{a}\langle x\rangle.P \mid a(x).Q \rightarrow P \mid Q \\
\textsc{Comm-C} & \bar{a}^{ch}\langle m\rangle.P \mid a^{ch}(n).Q \rightarrow P \mid Q[n{\leftarrow}m] \\
\textsc{Then} & \mathit{if}\ M = M\ \mathit{then}\ P\ \mathit{else}\ Q \rightarrow P \\
\textsc{Else} & \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q \rightarrow Q \\
 & \qquad (\text{when } \mathit{fv}(M,N) = \emptyset \text{ and } \mathcal{E} \nvdash M = N)
\end{array}$$

2.4 Frames

A *frame* is an extended process built up from active substitutions and the null process only. The *domain* $dom(\phi)$ of a frame ϕ is the set of variables upon which the active substitutions of ϕ act. The frame $\phi(P)$ of a process P is P in which every plain process embedded into P is set to $\mathbf{0}$. Similarly, the plain process $(P)^p$ associated with P is obtained by mapping every substitution over non-restricted variables to $\mathbf{0}$.

The following definitions are standard in the applied π-calculus:

- ϕ is *closed* when $\mathit{fv}(\phi) \subseteq dom(\phi)$;
- an evaluation context $C[\cdot]$ *closes* the frame ϕ when $C[\phi]$ is both well-formed and closed;
- two terms M and N are equal in the frame ϕ, written $\phi \vdash M = N$ when there exists a set of names $\boldsymbol{n}$ and a substitution σ (*i.e.* a public frame) such that $\phi \equiv \nu \boldsymbol{n}.\,\sigma$, $M\sigma = N\sigma$ and $\boldsymbol{n} \cap \mathit{fn}(M,N) = \emptyset$. Two terms are equal in the process P when they are equal in $\phi(P)$.

Definition 2.1 (Static equivalence). *Two closed frames ϕ and ψ are* statically equivalent*, written $\phi \approx_s \psi$, when $dom(\phi) = dom(\psi)$ and, for all terms M and N, $\phi \vdash M = N$ if and only if $\psi \vdash M = N$.*

Two processes are statically equivalent when their frames are.

As it is, static equivalence on non-closed frames does not lead to a congruence relation. For instance, with $\langle\,\cdot\,,\,\cdot\,\rangle$ being the pairing operation and π_1 the first projection, if we let $\phi = \nu n.\,\{^{\mathrm{dec}(\mathrm{enc}(\langle 1,n\rangle,1),y)}/_x\}$ and $\psi = \nu n.\,\{^{\mathrm{dec}(\mathrm{enc}(\langle 1,n\rangle,1),z)}/_x\}$ then $\phi \approx_s \psi$, but one can deduce $\pi_1(x) = 1$ from $\{^1/_y\} \mid \phi$ and not from $\{^1/_y\} \mid \psi$.

To overcome this issue, and later be able to write formulae characterizing static equivalence for both closed and non-closed frames, we introduce *strong* static equivalence $\approx_s^s$, which is the largest equivalence included in $\approx_s$ and closed by application of closing evaluation contexts. One can note that strong static equivalence is closed by application of arbitrary evaluation contexts (and not just closing ones), and that it coincides with static equivalence on closed frames.

2.5 Additional Operators

When splitting up a process P into a parallel composition of two subprocesses $P_1 \mid P_2$, two very different operations are performed, as both the dynamic and the static part of the process are split up into two extended processes. This does not match our intuition of local reasoning for processes. Indeed, consider a protocol $\nu n, n'.\, (\{^{\mathrm{ck}(n,n')}/_x\} \mid A(x,n) \mid B(x,n'))$ where Abelard and Héloïse share a compound key $\mathrm{ck}(n,n')$ generated from a nonce n of A and n' of B. Then a specification of the form $A \mid B$ would fail since this process is atomic. To overcome this situation we have broken down the parallel composition into two finer-grained operators: the first one $\nmid$ keeps the same frame while splitting up the plain process and, conversely, the second one $*$ keeps the same plain process while splitting up the frame.

Definition 2.2 (Process and frame compositions). *Given frames ϕ, ϕ_1, ϕ_2, plain processes P, P_1, P_2 and names $\boldsymbol{n_1}, \boldsymbol{n_2}$ such that $\boldsymbol{n_1} \cap fn(\phi_2, P_2) = \boldsymbol{n_2} \cap fn(\phi_1, P_1) = \emptyset$, we let*

$$\begin{aligned} \nu \boldsymbol{n_1}.\, ((\nu \boldsymbol{n_2}.\, \phi) \mid P_1) \nmid \nu \boldsymbol{n_2}.\, ((\nu \boldsymbol{n_1}.\, \phi) \mid P_2) &\triangleq \nu \boldsymbol{n_1 n_2}.\, (\phi \mid P_1 \mid P_2) \\ \nu \boldsymbol{n_1}.\, (\phi_1 \mid \nu \boldsymbol{n_2}.\, P) * \nu \boldsymbol{n_2}.\, (\phi_2 \mid \nu \boldsymbol{n_1}.\, P) &\triangleq \nu \boldsymbol{n_1 n_2}.\, (\phi_1 \mid \phi_2 \mid P)\,. \end{aligned}$$

In the following, for $\dagger$ in $\{\mid, \nmid, *\}$, we write $P \leftrightarrow P_1 \dagger P_2$ if there are P',P'_1,P'_2 such that $P \equiv P'$, $P_1 \equiv P'_1$, $P_2 \equiv P'_2$ and $P' = P'_1 \dagger P'_2$.

Remark 2.3. Formally, $P \leftrightarrow P_1 * P_2$ is a ternary relation, and for some P_1, P_2, one may have $P \leftrightarrow P_1 * P_2$, $P' \leftrightarrow P_1 * P_2$ for some non congruent P, P'. Albeit not a composition law, $*$ projects as a composition on frames: $\phi(P * Q) \equiv \phi(P) \mid \phi(Q)$. Ternary relations also arise in the relational models of BI, or in context logics.

3 A Spatial Logic for the Applied π-Calculus

3.1 Syntax and Semantics

We assume an infinite set $\mathcal{TV}$ of *term variables*, distinct from $\mathcal{V}$, ranged over with $t, t', \ldots$, and we write U, V for terms that possibly contain these term variables. We call $\mathcal{L}_{\text{spat}}$ the set of formulae defined by the following grammar:

$$\begin{aligned} A, B ::= {} & U = V \mid \neg A \mid A \wedge B \mid \Diamond A \mid \exists t.\, A \mid \mathit{И} u.\, A \mid \mathsf{H} u.\, A \mid A \oslash u \mid \textcircled{c} u \\ & \mid \mathbf{0} \mid A \nmid B \mid A \rhd B \mid \emptyset \mid A * B \mid A \mathrel{-\!\!*} B \end{aligned}$$

$U = V$ is the equality of terms w.r.t. the current frame, negation and conjunction are classical, and $\Diamond A$ is the strong reduction modality. $\exists t.\, A$ is term quantification and $\mathit{И} n.\, A$ and $\mathit{И} x.\, A$ are respectively the fresh name and variable quantifications. We use the same operator for both of these, as well as for the H, $\textcircled{c}$ and $\oslash$ operators, because our convention on namings lets us do so unambiguously. $\mathsf{H} u.\, A$ is the hidden name or variable quantification; $\textcircled{c} u$ means that u appears free in the process; $A \oslash u$ is hiding of name or variable u. $\mathbf{0}$ (resp. $\emptyset$) denotes the null plain process (resp. the empty frame) and $A \nmid B$ (reps. $A * B$) plain process (resp. frame) composition. $A \rhd B$ (resp. $A \mathrel{-\!\!*} B$) is a guarantee operator and the adjunct of $A \nmid B$ (resp. $A * B$).

We define two usual fragments of our logic: the extensional one, which lets one observe a process via its interactions with some (possibly constrained) environment, and the intensional one, which lets one explore the very structure of the process. We also distinguish between static operators, that only account for the frame, and dynamic ones, which account for the whole process. The four fragments are summed up by the table below, that defines which operators the formulae of each fragments may be composed of. The intensional fragment contains the extensional one and the dynamic one contains the static one; the static extensional fragment is thus common to all fragments, and the dynamic intensional one coincides with the whole logic.

	Static	*Dynamic*
Extensional	$=, \neg, \wedge, \exists, \mathcal{N}, \multimap, \oslash$	$\Diamond, \triangleright$
Intensional	$\mathsf{H}, *, \emptyset$	$\copyright, \mathbf{0}, \mathbin{!}$

$$
\begin{array}{lll}
P, v \vDash U = V & \Leftrightarrow & P \vdash Uv = Vv \\
P, v \vDash \neg A & \Leftrightarrow & P, v \nvDash A \\
P, v \vDash A_1 \wedge A_2 & \Leftrightarrow & P, v \vDash A_1 \text{ and } P, v \vDash A_2 \\
P, v \vDash \Diamond A & \Leftrightarrow & \exists P'.\, P \to P' \text{ and } P', v \vDash A \\
P, v \vDash \exists t.\, A & \Leftrightarrow & \exists M.\, P, (v\{t \rightarrow M\}) \vDash A \\
P, v \vDash \mathcal{N} u.\, A & \Leftrightarrow & \exists u' \notin fnv(P, v, A).\, P, v \vDash A[u \leftarrow u'] \\
P, v \vDash \mathsf{H} u.\, A & \Leftrightarrow & \exists u' \notin fnv(P, v, A).\, \exists P'.\, P \equiv \nu u'.\, P' \text{ and } P', v \vDash A[u \leftarrow u'] \\
P, v \vDash A \oslash n & \Leftrightarrow & \nu n.\, P, v \vDash A \\
P, v \vDash A \oslash x & \Leftrightarrow & x \in dom(P) \text{ and } \nu x.\, P, v \vDash A \\
P, v \vDash \copyright u & \Leftrightarrow & u \in \overline{fnv}(P) \\
P, v \vDash \mathbf{0} & \Leftrightarrow & (P)^p \equiv \mathbf{0} \\
P, v \vDash A_1 \mathbin{!} A_2 & \Leftrightarrow & \exists P_1, P_2.\, P \leftrightarrow P_1 \mathbin{!} P_2,\ P_1, v \vDash A_1 \text{ and } P_2, v \vDash A_2 \\
P, v \vDash A \triangleright B & \Leftrightarrow & \forall Q, R.\, (R \leftrightarrow P \mathbin{!} Q \text{ and } Q, v \vDash A) \text{ implies } R, v \vDash B \\
P, v \vDash \emptyset & \Leftrightarrow & \phi(P) \equiv \mathbf{0} \\
P, v \vDash A_1 * A_2 & \Leftrightarrow & \exists P_1, P_2.\, P \leftrightarrow P_1 * P_2,\ P_1, v \vDash A_1 \text{ and } P_2, v \vDash A_2 \\
P, v \vDash A \multimap B & \Leftrightarrow & \forall Q, R.\, (R \leftrightarrow P * Q \text{ and } Q, v \vDash A) \text{ implies } R, v \vDash B
\end{array}
$$

Fig. 1. Satisfaction relation

The operators' semantics, close to the one defined by Caires and Cardelli for the π-calculus [4], is given by a satisfaction relation described in Figure 1 whose judgements are of the form $P, v \vDash A$ between a process P, a spatial formula A and a valuation v. Valuations assign terms $\boldsymbol{M}$ to all the free variables $\boldsymbol{t}$ of the formula and are written $\{\boldsymbol{t} \rightarrow \boldsymbol{M}\}$ when $|\boldsymbol{t}| = |\boldsymbol{M}| = n$ and $v(t_i) = M_i$ for all $i \in \{1 \ldots n\}$. We write $v\{t \rightarrow M\}$ for the valuation v whose domain has been extended to t with $v(t) = M$. Finally, when A is closed and the valuation is empty, judgements are written $P \vDash A$. The following lemmas state that the logic behaves consistently w.r.t. structural congruence and that static operators may only state static properties indeed:

Lemma 3.1. *$P \vDash A$ and $P \equiv Q$ implies $Q \vDash A$.*

Lemma 3.2. *For every formula A of $\mathcal{L}^{stat}$, $P \vDash A$ iff $\phi(P) \vDash A$.*

Boolean operators are assumed to bind more tightly than compositions and adjunctions, which in turn bind more tightly than every other operator. Derived connectives $\forall t$, $\vee$, $\Leftrightarrow$ and $U \neq V$ are defined as usual, and so are the sets of free names and free variables of a formula A, written $fn(A)$ and $fv(A)$.

3.2 Derived Formulae

The formulae below will be useful in the following sections:

$$\top \triangleq \mathbf{0} \vee \neg\mathbf{0} \qquad \bot \triangleq \neg\top \qquad A[B] \triangleq (A \wedge \mathbf{0}) \mathbin{¦} ((B \wedge \emptyset) * \top)$$

$$A \blacktriangleright B \triangleq \neg(A \triangleright \neg B) \qquad A \mathrel{-\!\!\circledast} B \triangleq \neg(A \mathrel{-\!\!*} \neg B) \qquad \mathbf{1} \triangleq \neg\mathbf{0} \wedge \neg(\neg\mathbf{0} \mathbin{¦} \neg\mathbf{0})$$

$$\mathbb{I} \triangleq \neg\emptyset \wedge \neg(\neg\emptyset * \neg\emptyset) \qquad \mathsf{public} \triangleq \neg \mathsf{H} n.\, \copyright n \qquad \mathsf{single} \triangleq \mathbf{1} \wedge \emptyset \wedge \mathsf{public}$$

Their meanings for a process P is that there must exist a process $Q \equiv P$ such that:

- $\top$: nothing is required; $\bot$ is always false;
- $A[B]$: $\phi(Q)$ verifies A and $(Q)^p$ verifies B;
- $A \blacktriangleright B$ (this is the dual of $\triangleright$): there are Q', R such that $R \leftrightarrow Q \mathbin{¦} Q'$, $Q' \vDash A$ and $R \vDash B$, and similarly for $-\!\!\circledast$;
- $\mathbf{1}$ (resp. $\mathbb{I}$): $(Q)^p$ (resp. $\phi(Q)$) is not null and cannot be divided into two non-null processes;
- public: Q has no bound name: $\forall n, Q'.\, Q \equiv \nu n.\, Q' \Rightarrow n \notin \overline{fn}(Q')$;
- single: Q is guarded, either by a communication or by a conditional construct.

3.3 Cryptographic Examples

We now propose, on a very basic example, some possible avenues for using the spatial logic for the specification of some cryptographic properties. As usual, we interpret the frame as the history of past communications: restricted names are nonces or secrets, and each active substitution holds the content of an emitted message. Recall the frame $\nu n, s.\, \Phi$ of the introduction, modeling a situation where an encrypted secret s had been transmitted using a published public key $\mathrm{pk}(n)$ — we assume here the equational theory axiom $\mathrm{dec}(\mathrm{enc}(x, \mathrm{pk}(y)), \mathrm{sk}(y)) = x$ — and consider the frame $\nu n, s.\, \Phi \mid \phi = \nu n, s.\, \{^{\mathrm{enc}(s,y)}/_x\} \mid \{^{\mathrm{pk}(n)}/_y\} \mid \phi$.

Following the definition of the applied π-calculus, we will say that the secret s is deducible from this frame if the formula $\mathsf{leak} \triangleq \exists t.\, x = \mathrm{enc}(t, y)$ holds; for instance, choosing $\phi = \{^{\mathrm{sk}(n)}/_z\}$ would yield such a leak, with witness $t = \mathrm{dec}(x, z)$. The formula $\exists t. \forall t'. \mathsf{H} n.\, (t = \mathrm{pk}(n) \wedge t' \neq \mathrm{sk}(n))$ asserts that the published key is indeed public, whereas its associated private key is secret. An emitted message M represented by an active substitution $\{^M/_z\}$ in ϕ is part of the cause for a leak if the formula $(\neg\mathsf{leak}) \oslash z \wedge \mathsf{leak}$ holds.

One could also express static properties about authenticated sessions: in a protocol where each user is assigned a session identifier (here, a secret name) used in every subsequent communication, one may count the number of opened sessions with $*$-conjunctions

of the $\mathbb{I}$ formula. Indeed, if every other nonce used by the protocol within a session is generated from the session identifier, each subframe verifying $\mathbb{I}$ will correspond to a different session.

The dynamic part of the logic allows one to reason about the execution in isolation of some partners of a given protocol, or in a context which abides by some policy of the protocol: formulae $\mathsf{Client} \mid \mathsf{Server}$, or $\mathsf{Client} \triangleright \mathsf{Attack}$, would describe a protocol with a client and a server, or a server that might be attacked by a context following the specification of a genuine client.

4 Spatial Logic Applied to Frames

In this section, we establish that logical equivalences induced by the static fragments match static equivalences that have originally been proposed for the applied π-calculus: structural congruence for frames for the intensional fragment, and static equivalence for the extensional fragment.

4.1 Intensional Characterization

The formula $\mathsf{Subst}(x = M)$ below characterizes processes of the form $\{^{M}/_{x}\}$, for a given x and a given M. The other two will be useful for our quantifier elimination procedure in Section 6.

$$\mathsf{public\_frame} \triangleq \neg(\top * (\mathbb{I} \wedge \mathsf{H}x.\,\mathbb{I})) \qquad \mathsf{Subst}(x) \triangleq \mathsf{public\_frame} \wedge (\emptyset \oslash x)$$

$$\mathsf{Subst}(x = M) \triangleq \emptyset \oslash x \wedge x = M$$

Lemma 4.1. *For each process P, variable x and term M such that $\mathcal{E} \not\vdash x = M$, we have:*

- $P \vDash \mathsf{public\_frame}$ *iff* $\phi(P) \equiv \{^{M}/_{\boldsymbol{x}}\}$ *for some variables* $\boldsymbol{x}$ *and terms* $\boldsymbol{M}$*;*
- $P \vDash \mathsf{Subst}(x)$ *iff* $\phi(P) \equiv \{^{N}/_{x}\}$ *for some* N*;*
- $P \vDash \mathsf{Subst}(x = M)$ *iff* $\phi(P) \equiv \{^{M}/_{x}\}$.

Proof: First, observe that if $P \vDash \mathbb{I} \wedge \mathsf{H}x.\,\mathbb{I}$ then $P \vDash \mathbb{I}$, so either $\phi(P)$ is a single public active substitution or $\phi(P) \equiv \nu\boldsymbol{n}.\,\sigma$ where σ cannot be split into σ_1 and σ_2 that do not share names in $\boldsymbol{n}$. Moreover, $P \vDash \mathsf{H}x.\,\mathbb{I}$ so we can reveal a fresh variable to obtain a process whose frame still verifies $\mathbb{I}$. This is only possible if P's frame was of the latter form and if the active substitution created by this revelation makes use of some restricted name in $\boldsymbol{n}$. This illustrates one peculiar behaviour of variable revelation: it may reveal a substitution under the scope of an arbitrary number of name restrictions. Now, if $P \vDash \mathsf{public\_frame}$ then we cannot isolate a non-public subframe of P, so P's frame is public.

Reciprocally, if $\phi(P)$ is not public, then there is n such that $\phi \equiv (\nu n.\,\phi_1) * \phi_2$, $n \in \overline{fn}(\phi_1)$ and $\nu n.\,\phi_1 \vDash \mathbb{I}$. Then, for $x \notin dom(P)$, we have $\nu n.\,\phi_1 \equiv \nu x, n.\,(\phi_1 \mid \{^{n}/_{x}\})$ so $\nu n.\,\phi_1 \vDash \mathsf{H}x.\,\mathbb{I}$, hence the result.

The other two formulae are straightforward. Observe that the hide operator is used in conjunction with the $\emptyset$ predicate to state both $x \in dom(P)$ and $dom(P) \subseteq \{x\}$. $\square$

Once this basic block is defined, one can easily build up a formula capturing processes in a certain structural congruence class, as expressed by the following theorem:

Theorem 4.2 (Characteristic formulae for frames). *For all frames ϕ there exists a formula F_ϕ in $\mathcal{L}^{stat}_{int}$ such that for all extended processes P, $P \vDash \mathsf{F}_\phi$ if and only if $\phi(P) \equiv \phi$.*

For example, a characteristic formula for $\nu n, s.\, \Phi$ is $(\mathsf{H}s.\, \mathsf{Subst}(x = \mathrm{enc}(s, y))) * (\mathsf{H}n.\, \mathsf{Subst}(y = \mathrm{pk}(n)))$. Together with Lemma 3.1, this theorem gives a precise definition of logical equivalence induced by $\mathcal{L}^{\mathrm{stat}}_{\mathrm{int}}$ on frames:

Corollary 4.3 (Logical equivalence in $\mathcal{L}^{\mathrm{stat}}_{\mathrm{int}}$). *For all extended processes P and Q, P and Q satisfy the same formulae of $\mathcal{L}^{stat}_{int}$ if and only if $\phi(P) \equiv \phi(Q)$.*

4.2 Extensional Characterization

We will show in this section that logical equivalence for $\mathcal{L}^{\mathrm{stat}}_{\mathrm{ext}}$, or extensional equivalence, coincides with strong static equivalence and that, given a closed frame ϕ, one can construct a formula $\mathsf{F}^{\approx_s}_\phi$ characterizing the equivalence class of ϕ. The right-to-left inclusion is given by the following lemma:

Lemma 4.4. *$\phi \approx^s_s \psi$ and $\phi \vDash A$ implies $\psi \vDash A$.*

For the converse of the above lemma, let us first remark that one can characterize frames whose domains are $\boldsymbol{x}$ using the formula $\emptyset \oslash \boldsymbol{x}$. Then, one can define a characteristic formula $\mathsf{F}^{\approx_s}_\sigma$ for a public frame $\sigma = \{{}^{M}/_{\boldsymbol{x}}\}$ of size n:

$$\mathsf{F}^{\approx_s}_\sigma \triangleq \emptyset \oslash \boldsymbol{x} \wedge \bigwedge_{i=1}^{n} x_i = M_i\,.$$

Let ϕ be a closed frame $\nu \boldsymbol{n}.\, \sigma$ with σ a public frame, and consider the formula

$$\phi \text{ forces } U = V \triangleq \mathsf{F}^{\approx_s}_\sigma \mathrel{-\!\!*} \big((U = V) \oslash \boldsymbol{n}\big)\,.$$

Then $\emptyset, v \vDash \phi$ forces $U = V$ if and only if $\phi, v \vDash U = V$. Moreover, one can internalize an assumption $\emptyset \vDash A$ in the logic: a process P satisfies $(\emptyset \wedge \neg A) \mathrel{-\!\!*} \bot$ if and only if $\emptyset \vDash A$. We may then derive characteristic formulae for static equivalence on closed frames:

$$\mathsf{F}^{\approx_s}_\phi \triangleq \emptyset \oslash \boldsymbol{x} \wedge \forall t, t'.\, \big((\emptyset \wedge \neg\phi \text{ forces } t = t') \mathrel{-\!\!*} \bot\big) \Leftrightarrow t = t'\,.$$

Theorem 4.5 (Formulae for static equivalence). *For all closed frames ϕ, ψ, $\psi \vDash \mathsf{F}^{\approx_s}_\phi$ if and only if $\phi \approx_s \psi$.*

Using this theorem and the Lemma above, one concludes that two closed frames ϕ and ψ satisfy the same formulae of $\mathcal{L}^{\mathrm{stat}}_{\mathrm{ext}}$ if and only if $\phi \approx_s \psi$.

For the general case, let us consider two non-closed, logically equivalent frames ϕ_1 and ϕ_2. Then, for any closing evaluation context $C \equiv \nu \boldsymbol{n}.\, (\,\cdot \mid \sigma)$, they should both satisfy the formula $\mathsf{F}_\sigma \mathrel{-\!\!*} \mathsf{F}^{\approx_s}_{C[\phi_1]} \oslash \boldsymbol{n}$. Thus, for all closing evaluation contexts C, $C[\phi_2] \vDash \mathsf{F}^{\approx_s}_{C[\phi_1]}$ so $C[\phi_2] \approx_s C[\phi_1]$. This shows $\phi_1 \approx^s_s \phi_2$, so logical equivalence

on frames is indeed strong static equivalence on frames (and thus on processes, by Lemma 3.2).

5 Logical Characterization of Processes

In this section we study the logical equivalence induced by the dynamic intensional fragment.[1] More precisely, we write $P =_L Q$ if P, Q cannot be discriminated by $\mathcal{L}_{\text{spat}}$ formulae and look for a better understanding of $=_L$. We introduce a notion of intensional bisimulation that aims at characterizing $=_L$ by an Ehrenfeucht-Fraïssé game.

Definition 5.1 (Intensional bisimulation). *A relation $\mathcal{R}$ is an intensional bisimulation if $\mathcal{R}$ is symmetric and the following assertions hold for all $(P, Q) \in \mathcal{R}$:*

1. $\phi(P) \equiv \phi(Q)$
2. *if* $P^p \equiv \mathbf{0}$, *then* $Q^p \equiv \mathbf{0}$;
3. *if* $u \in \overline{fnv}(P)$, *then* $u \in \overline{fnv}(Q)$;
4. *if there is* P' *s.t.* $P \equiv \nu u.\, P'$, *then there is* Q' *s.t.* $Q \equiv \nu u.\, Q'$ *and* $P' \mathcal{R} Q'$;
5. *for* $\dagger \in \{*, |\}$, *for all* P_1, P_2, *if* $P \leftrightarrow P_1 \dagger P_2$, *then there are* Q_1, Q_2 *such that* $Q \leftrightarrow Q_1 \dagger Q_2$ *and* $P_i \mathcal{R} Q_i$;
6. *for* $\dagger \in \{*, |\}$, *for all* P_1, P', *if* $P_1 \leftrightarrow P \dagger P'$, *then there are* Q_1, Q' *such that* $Q_1 \leftrightarrow Q \dagger Q'$, $P' \mathcal{R} Q'$ *and* $P_1 \mathcal{R} Q_1$;
7. *if there is* P' *s.t.* $P \rightarrow P'$, *then there is* Q' *s.t.* $Q \rightarrow Q'$ *and* $P' \mathcal{R} Q'$;
8. $\nu u.\, P \mathcal{R} \nu u.\, Q$.

Let us stress the fact that the equivalents of conditions 6 and 8 do not occur in the original intensional bisimulation [14]. Fortunately, in that case the intensional bisimilarity was a congruence, and as a consequence, conditions 6 and 8 were admissible. Note moreover that conditions 6 and 8 do not entail that $\mathcal{R}$ is a congruence (even with $\dagger = |$).

Proposition 5.2. *Let $\mathcal{R}$ be an intensional bisimulation. Then $\mathcal{R} \subseteq =_L$.*

We now give two examples of intensional bisimulations that illustrate that logical equivalence is strictly coarser than structural congruence, and is not even a congruence in general. These bisimulations are based on the notion of shift functions. A unary function symbol f is called a shift function if there are unary function symbols $g_1, \ldots, g_n$ such that $\mathcal{E} \vdash f(\boldsymbol{g}(x)) = \boldsymbol{g}(f(x)) = x$. In the remainder, we assume some fixed shift function f — we will later on consider the case of the equational theory of trees, for which there is no such function. In cryptographic terms, M and $f(M)$ represent two different pieces of information that are deducible one from another by a linear deduction, which explains why they may be indistinguishable for some notion of observer matching our logic.

Let a be some fixed channel name, f a shift function and $\boldsymbol{g}$ such that $\mathcal{E} \vdash f(\boldsymbol{g}(x)) = \boldsymbol{g}(f(x)) = x$. We consider a transformation $\mathsf{shift}_a^f(P)$ that intuitively shifts all term communications of P on channel a using function f — this could be thought of as a reversible noise introduced globally on all communications over a.

[1] We also observed that dynamic extensional fragment does not characterize behavioral equivalence, but due to lack of space we will not develop this point.

Definition 5.2 (Shifted channels). *The transformation* $\mathsf{shift}_a^f(.)$ *is inductively defined as a morphism for all syntactic operators but term inputs and outputs on* a*, for which it is defined as follows:*

$$\mathsf{shift}_a^f(\bar{a}\langle M\rangle.P) \triangleq \bar{a}\langle f(M)\rangle.\mathsf{shift}_a^f(P)$$
$$\mathsf{shift}_a^f(a(x).P) \triangleq a(x).\mathsf{shift}_a^f(P[x{\leftarrow}\boldsymbol{g}(x)]).$$

Proposition 5.4. *The symmetric closure of* $\mathcal{R} = \{(P, \mathsf{shift}_f^a(P)), P \in \mathcal{P}\}$ *is an intensional bisimulation.*

Propositions 5.2 and 5.4 have some quite unexpected consequences on logical equivalence. First, it entails the following equivalences:

$$\bar{a}\langle 0\rangle =_L \bar{a}\langle f(0)\rangle \text{ and } \bar{a}\langle 0\rangle \,|\, \bar{b}\langle 0\rangle =_L \bar{a}\langle f(0)\rangle \,|\, \bar{b}\langle 0\rangle.$$

But the noise introduced on a channel should affect *all* of its communications, as it could otherwise be observed; in particular, it can be proved that:

$$\bar{a}\langle 0\rangle \,|\, \bar{a}\langle 0\rangle \;\neg_L\; \bar{a}\langle f(0)\rangle \,|\, \bar{a}\langle 0\rangle,$$

which shows that $=_L$ is not a congruence. Such a phenomenon was already observed for the spatial logic of CCS [15], where $=_L$ coincided with structural congruence up to injective renaming. Non-congruence makes the proof of the converse of Proposition 5.2 much harder than the Howe-like method used e.g. by Sangiorgi [14], even in the very simple case of CCS. Indeed, a global quantification over the shift function used for each channel should be expressed at the logical level, which calls on for a quantifiers elimination result; despite some progress in that direction (see next section), we did not succeed in using them to prove that $=_L$ is an intensional bisimulation.

Let now $\sim$ be the smallest equivalence on pairs of terms $M = N$ such that:

$$\begin{array}{ll} \textsc{Symmetry} & M = N \sim N = M \\ \textsc{Shift} & M = N \sim f(M) = f(N) \end{array}$$

where f is a shift function. Let moreover $\equiv'$ be the smallest congruence extending $\equiv$ with the following axiom:

$$\textsc{Test} \qquad \begin{array}{l} \mathit{if}\ \mathit{test} \\ \quad \mathit{then}\ P\ \mathit{else}\ Q \end{array} \equiv' \begin{array}{l} \mathit{if}\ \mathit{test}' \\ \quad \mathit{then}\ P\ \mathit{else}\ Q \end{array}$$

when $\mathit{test} \sim \mathit{test}'$. Then the following result can be established:

Proposition 5.5. $\equiv'$ *is an intensional bisimulation.*

As mentioned above, we did not succeed to prove any completeness result in the general case, but we managed to derive some widget formulae that are sufficiently expressive to characterize $\equiv'$ on, at least, the equational theory of finite trees. Due to lack of space, we skip the quite involved construction.

Theorem 5.6. *Let* $\mathcal{E}$ *be the theory of finite trees. Then for every process* P *there is a formula* $\mathsf{F}_P \in \mathcal{L}$ *such that for all processes* Q*,* $Q \models \mathsf{F}_P$ *if and only if* $Q \equiv' P$*. In particular,* $\equiv'$ *is the same as* $=_L$*.*

On π-calculus processes, *i.e.* processes that contain neither term communications nor conditionals, it can be shown using a similar technique that logical equivalence is structural congruence for all equational theories.

6 Elimination of Term Quantification

This section is devoted to the construction of a translation of any formula A of $\mathcal{L}_{\text{spat}}$ into a logically equivalent one that does not make use of term quantification, thus proving the following theorem:

Theorem 6.1 (Term quantification elimination). *For every closed formula $A \in \mathcal{L}$, there is a formula $[\![A]\!] \in \mathcal{L} \setminus \{\exists\}$ such that $A \Leftrightarrow [\![A]\!]$ is valid.*

$[\![A]\!]$ is defined by structural induction on the formula A. It leaves most of A's structure unchanged, while replacing every subformula $\exists t.\, A'$ with a formula of the form $\mathsf{H}x.\, [\![A']\!]_{\{t \to x\}}$. Hence, the H quantifier is in charge of picking a term M for the new active substitution $\{^M/_x\}$ it reveals, thus mimicking term quantification. Further occurrences of t in the formula will have to be replaced by x. This inductively builds up an *environment frame* placed alongside the actual process that records witnesses of term quantifications, but for which some maintenance work is needed during the translation of a formula. For instance, we will need to copy this environment on each side of a $*$ operator, and on the left-hand side of a $-\!\!*$. Moreover, to follow the semantics of $\exists t.\, A$, one has to make sure that this substitution does not use any hidden name of the process or any of the substitutions belonging to the environment frame.

To keep track of this, the translation will have to be of the form $[\![A]\!]_v$ where v is a valuation $\{\boldsymbol{t} \to \boldsymbol{x}\}$ that lets previously encountered term variables point to their corresponding variables in the domain of the environment frame. The translation thus starts with an empty valuation: $[\![A]\!] \triangleq [\![A]\!]_\emptyset$, and the valuation grows up each time a term quantification is encountered. We write e for the environment $\{\boldsymbol{x} \to \boldsymbol{M}\}$ corresponding to the environment frame $[\![e]\!] \triangleq \{^{\boldsymbol{M}}/_{\boldsymbol{x}}\}$. Moreover, we only consider environments e and translations $[\![A]\!]_v$ where $fv(A, \boldsymbol{M}) \cap \boldsymbol{x} = \emptyset$. Finally, when the domain of e matches the codomain of v, we write $e \circ v$ for the valuation $\{\boldsymbol{t} \to \boldsymbol{M}\}$.

We are now ready to give the inductive lemma we want to prove on $[\![A]\!]_v$:

Lemma 6.2 (Inductive hypothesis). *$P \vDash [\![A]\!]_v$ if and only if there exists Q and e such that $P \equiv Q \,|\, [\![e]\!]$, $fv(Q) \cap dom([\![e]\!]) = \emptyset$, and $Q, e \circ v \vDash A$.*

To meet the requirements of this lemma and make sure that P is indeed the composition of a process Q and an environment frame corresponding to e, we first define a formula Φ_v that will have to be verified at every step of the translation:

$$\Phi_{\{\boldsymbol{t} \to \boldsymbol{x}\}} \triangleq \bigwedge_{x \in \boldsymbol{x}} (\mathsf{Subst}(x) * \neg \copyright x)\,.$$

Lemma 6.3. *For all processes P and valuations v, $P \vDash \Phi_v$ if and only if there exists a process Q and an environment e such that $P \equiv Q \,|\, [\![e]\!]$ and $fv(Q) \cap dom([\![e]\!]) = \emptyset$.*

The translation of all the operators of the logic can be found in the companion technical report [16]. We will give here the proof sketches for the translations of $\exists t.\, A$ and $A_1 * A_2$. The actual translation of term quantification is as follows, where $x_{n+1} \notin fv(A, v)$:

$$[\![\exists t.\, A]\!]_v \triangleq \Phi_v \wedge \mathsf{H}x_{n+1}.\, [\![A]\!]_{v\{t \to x_{n+1}\}}\,.$$

It merely creates a fresh substitution, as the inductive hypothesis on $[\![A]\!]_{v\{t \to x_{n+1}\}}$ suffices to enforce the validity of the new environment frame.

The translation of frame composition needs the valuation frame to be copied in order for it to be present alongside both subprocesses. It is performed as follows:

$$[\![A_1 * A_2]\!]_v \triangleq \Phi_v \wedge \mathit{И}\boldsymbol{x'}.\, (\Phi_{v'} \wedge \bigwedge_{x \in \boldsymbol{x}} \neg ⓒ x \wedge \mathsf{Subst}(\boldsymbol{x'})) \multimap\!\circledast \begin{pmatrix} *_{i=1}^{n} (\mathsf{Subst}(x_i, x'_i) \wedge x'_i = x_i) * \top \\ \wedge [\![A_1]\!]_v * [\![A_2]\!]_{v'} \end{pmatrix}.$$

The idea is to add a new environment frame over fresh variables $\boldsymbol{x'}$. The left-hand side of $\multimap\!\circledast$ ensures that this is a valid environment which does not make use of the variables of the previous environment. This is to avoid the possibility of creating active substitutions of the form $\{^{x}/_{x'}\}$ which would not make sense once we separate them from the first environment. The right-hand side makes sure that both environments are the same and distributes them over the interpretations of sub-formulae A_1 and A_2.

7 Conclusion

Related work. Spatial logics for process algebrae with explicit resources have been first studied by Pym [17]. The idea of distributing assertions about knowledge in space using spatial logics has been explored by Mardare [18]. More examples of applications of spatial connectives in cryptographic logics can be found in Kramer's thesis [19]. Hüttel *et al.* gave a logical characterization and characteristic formulae for static equivalence [13] for some classes of equational theories.

Extensions. One natural way to extend the logic could be to consider a weak, several steps version of the $\diamondsuit$ modality. We conjecture that this would allow us to handle the full applied π-calculus with replication, as in the case of ambients [6].

Future work. The decidability status of the logic depends on the considered equational theory, and is already limited by strong undecidability results for the first-order equational logic. At the time of this writing, we are investigating the decidability of the model-checking problem for (fragments of) our logic. A first positive result has been obtained for the static part of the logic without the magic wand operator or the ability to reveal variables, and for a very restricted class of equational theories [16]. A promising line of work would be to try to extend this result to common equational theories, and to allow the use of variable revelation in formulae.

Acknowledgments. We acknowledge, among others, Steve Kremer, Ralf Treinen and Simon Kramer for valuable discussions.

References

1. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 55–74 (2002)
2. Ishtiaq, S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL 2001, pp. 14–26 (2001)

3. Gordon, A., Cardelli, L.: Anytime, anywhere: Modal logics for mobile ambients. In: Press, A. (ed.) POPL 2000, pp. 365–377 (2000)
4. Caires, L., Cardelli, L.: A spatial logic for concurrency (part I). Journal of Information and Computation 186(2) (2003)
5. Hirschkoff, D., Lozes, É., Sangiorgi, D.: Minimality results for spatial logics. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 252–264. Springer, Heidelberg (2003)
6. Hirschkoff, D., Lozes, É., Sangiorgi, D.: On the expressiveness of the ambient logic. Logical Methods in Computer Science 2(2) (March 2006)
7. Hirschkoff, D.: An extensional spatial logic for mobile processes. In: Jin, H., Pan, Y., Xiao, N., Sun, J. (eds.) CONCUR 2002. LNCS, vol. 3252. Springer, Heidelberg (2002)
8. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL 2001, pp. 104–115 (2001)
9. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i. Inf. Comput. 100(1), 1–40 (1992)
10. Dawar, A., Gardner, P., Ghelli, G.: Expressiveness and complexity of graph logic. Inf. Comput. 205(3), 263–310 (2007)
11. Calcagno, C., Gardner, P., Hague, M.: From separation logic to first-order logic. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 395–409. Springer, Heidelberg (2005)
12. Kuncak, V., Rinard, M.: On spatial conjunction as second-order logic. Technical report, MIT CSAIL (October 2004)
13. Hüttel, H., Pedersen, M.D.: A logical characterisation of static equivalence. Electron. Notes Theor. Comput. Sci. 173, 139–157 (2007)
14. Sangiorgi, D.: Extensionality and intensionality of the ambient logics. In: POPL (2001)
15. Caires, L., Lozes, É.: Elimination of quantifiers and undecidability in spatial logics for concurrency. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 240–257. Springer, Heidelberg (2004)
16. Villard, J., Lozes, É., Treinen, R.: A spatial equational logic for the applied pi-calculus. Research Report LSV-08-10, LSV, ENS Cachan, France, 44 pages (March 2008)
17. Pym, D., Tofts, C.: A Calculus and logic of resources and processes. Formal Aspects of Computing 18(4), 495–517 (2006)
18. Mardare, R.: Observing distributed computation. a dynamic-epistemic approach. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 379–393. Springer, Heidelberg (2007)
19. Kramer, S.: Logical Concepts in Cryptography. PhD thesis, École Polytechnique Fédérale de Lausanne (2007)

Structured Interactional Exceptions in Session Types

Marco Carbone[1], Kohei Honda[1], and Nobuko Yoshida[2]

[1] Queen Mary, University of London
[2] Imperial College London

Abstract. We propose an interactional generalisation of structured exceptions based on the session type discipline. Interactional exceptions allow communicating peers to asynchronously and collaboratively escape from the middle of a dialogue and reach another in a coordinated fashion, under an arbitrary nesting of exceptions. New exception types guarantee communication safety and offer a precise type-abstraction of advanced conversation patterns found in practice. Protocols for coordinating normal and exceptional exit among asynchronously running sessions are introduced. The liveness property established under these protocols guarantees consistency of coordinated exception handling among communicating peers.

1 Introduction

Structured exceptions in modern programming languages such as Java and C♯ allow a thread of control in a block (often designated as "`try` block") to get transferred to another block (exception handler, "`catch` block"), when a system or user raises an event called *exception*. Their central merit is to enable a dynamic escape from a block of code to another (like `goto`), but in a controlled and structured way (unlike `goto`). They are useful not only for error-handling but more generally for a flexible control flow while preserving well-structured description and type-safety.

This paper studies the new notion of structured exceptions for distributed, concurrent, asynchronously communicating programs based on session types [10,17], motivated by collaboration with industry partners in web services [19] and financial protocols [14]. These two application domains contain a wealth of structured conversation patterns arising from practical needs [11], and many of these patterns crucially rely on dynamic escape: a conversation is interrupted by a special communication action, after which all peers move to a different stage. Realising such conversation patterns requires a consistent propagation of exception messages among concurrently communicating peers; an exception affects not only a sequential thread but also a collection of parallel processes; and an escape needs to move into another dialogue in a concerted manner. The distinguishing feature of these exceptions in comparison with their traditional counterpart is that they demand not only local but also coordinated actions among communicating peers. We call such exceptions, *interactional exceptions*.

As a simple example of interactional exceptions, we present the following scenario, coming from widely used financial protocols. Henceforth we assume a message passing in a session is asynchronous, i.e. the completion of a sending action does not need a

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 402–417, 2008.

handshake with its receiver, a standard assumption in financial messaging [1]. Suppose Seller wishes to sell a product to Buyer.

1. Seller repeats sending quotes of a product without waiting for an acknowledgement;
2. When Buyer replies with his interest in one of the quotes, the loop terminates and Seller and Buyer move to another stage, for e.g. completion of the transaction.

This simple conversation pattern contains an asynchronous escape from one part of a conversation to another. After one party aborts, the same thing should happen to the other, both moving together to another part of the conversation.

As a second example, we continue the above scenario, extending to the situation where Buyer and Seller negotiate the price of the product through Broker.

1. Buyer initiates a conversation (session) with Broker, in order to buy a product.
2. As a result, Broker initiates a conversation with Seller, and starts brokering between Buyer and Seller, to reach a successful transaction.
3. If an exceptional circumstance arises between 1 and 2 (e.g. a legal issue), Buyer or Broker will abort and they together move to an exception dialogue to quit the transactions formally.
4. On the other hand, if there is an exceptional circumstance during 2, then there is an exception dialogue involving all of Broker, Seller and Buyer.

Above, an exception handling at Broker is *nested*, whose later, or inner, exception handling (4, involving all three parties) supersedes the earlier, or outer, one (3, involving only Broker and Seller). As a conversation evolves, more communication peers may be involved, making it necessary to coordinate more parties when an exception is raised.

To maintain the virtues of traditional structured exceptions, as well as those of the existing session types discipline, we may as well demand the following three properties for this generalised form of exceptions.

- *flexibility*: it should allow asynchronous escape at any desired point of a conversation, including nested exceptions;
- *consistency*: even under asynchrony, messages in a "default" conversation should not get mixed up with those in an "exception" conversation, under arbitrary nesting;
- *type safety*: communications inside a session always take place linearly and without communication mismatch, carrying out fundamental properties of foregoing session type disciplines.

We address these requirements by extending session types with the following features:

1. Asynchronous exceptions where nested scopes are consistently handled by a meta-level reduction and a stack discipline. A simple machinery, based on exception levels, prevents mix-up of messages in normal and exception conversations.
2. An operational structure for coordinating exceptions including the protocols to propagate exceptions, to handle normal and exceptional exit from a conversation, and to coordinate entry into exception-handling conversations.
3. A type structure for interactional exceptions which minimally extends that of the existing session types. They ensure communication safety and liveness, which together guarantee the consistency of the introduced operational structures.

The stipulated formal semantics of interactional exceptions is intended to suggest a possible framework of implementation, as discussed in § 5. As far as we know, this work is the first to present a consistent extension of the session types discipline to interactional exceptions, backed up by its key formal properties.

2 Session Calculus with Interactional Exceptions

Syntax. We introduce the syntax of processes using the π-calculus with session primitives [10]. Let a, b range over *service channels*; s, r, t over *session channels*; x, y, z over *variables*; and $X, X', \ldots$ over *term variables*. We first introduce the syntax of processes ($P, Q, R, \ldots$) written by programmers.

$$
\begin{array}{lrlr}
P ::= & * c(\lambda)[P,Q] & \text{(service)} & \mid \overline{c}(\lambda)[\tilde{\kappa},P,Q] \quad \text{(request)} \\
& \mid \kappa?(x).\, P & \text{(input)} & \mid \kappa!\langle e\rangle.\, P \quad \text{(output)} \\
& \mid \kappa \triangleright \{l_i : P_i\}_{i\in I} & \text{(branch)} & \mid \kappa \triangleleft l.\, P \quad \text{(select)} \\
& \mid P \mid Q & \text{(par)} & \mid \textbf{if } e \textbf{ then } P \textbf{ else } P \quad \text{(cond)} \\
& \mid \mathbf{0} & \text{(inact)} & \mid (\nu a)\, P \quad \text{(resServ)} \\
& \mid X & \text{(termVar)} & \mid \mu X.\, P \quad \text{(recursion)} \\
& \mid \textbf{throw} & \text{(throw)} &
\end{array}
$$

$$
e ::= a \mid \mathsf{tt} \mid \mathsf{ff} \mid e \text{ and } e \mid \neg e \mid \ldots \qquad c ::= a \mid x \qquad \kappa, \lambda ::= s^p
$$

Session channel s^p is a *polarised* channel [8] where variable p ranges over polarities $\{+, -\}$. We define the dual of a polarised channel s^p as $\overline{s^+} = s^-$ and $\overline{s^-} = s^+$.

A service $*a(\lambda)[P, Q]$, named a, is a replicated process where P is the *default process* and Q is the *exception handler*. By replication a service is always available (following Service Channel Principle (SCP): "services should always be available in multiple copies" [6], like services at URLs). When $*a(\lambda)[P, Q]$ is requested for a session via a shared name a, a fresh session channel is established, and through this channel P is engaged in a series of communication actions, possibly followed by Q if an exception takes place. A request $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$ interacts with a service via c and establishes a fresh session λ, with its *default process* P and *handler* Q. The channels $\tilde{\kappa}$ are the already established sessions with which the handler Q gets associated with, thus allowing nesting of exceptions. We also let λ itself be included in $\tilde{\kappa}$, which is convenient for typing. We call $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$ a *refinement* of each κ_i for $\kappa_i \in \{\tilde{\kappa}\} \setminus \{\lambda\}$, since its handler Q refines the handlers of the previous sessions enabling nested exceptions. **throw** is a process which is about to throw an exception. All other constructs are from [6,10].

Free/bound (term) variables/channels and α-equivalence are standard. $\mathsf{fsc}(P)$, $\mathsf{fn}(P)$ and $\mathsf{fv}(P)$ respectively denote the sets of free session channels, service channels, and variables in P. We call *program* a process which does not contain free variables or free session channels. We often omit the tailing $\mathbf{0}$.

For having consistent operational semantics, we stipulate the following syntactic constraints: (i) recursions should be *guarded*, i.e. P in $\mu X.\, P$ is prefixed by an input/output/branch/select/conditional; (ii) a service can never occur under an input/output/recursion prefix nor inside a default process or a handler thus protecting the availability of services from exceptions; and (iii) in $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$, a free term variable never occurs in

P or Q and, for each $\overline{c'}(\lambda')[\tilde{\kappa}', P', Q']$ occurring in P, we have $\kappa_i \in \tilde{\kappa}'$ if and only if $\tilde{\kappa} \subseteq \tilde{\kappa}'$ (for consistent exception propagation). Further, such a refinement never occurs inside a handler (otherwise we have ambiguity when launching a handler).

In the present paper we also stipulate that **throw** never occurs inside a handler. This prevents a handler from throwing a further exception in the same session. We do not consider such "cascading exceptions" (which is another kind of nested exceptions) for the sake of simpler presentation: their treatment is discussed in §5.

Example 1 (Asynchronous Escape). We can write the first example in § 1 as:

$$\begin{array}{ll} \texttt{Buyer} = \overline{\texttt{chSeller}}(s^+)[\ s^+, & \texttt{Seller} = *\texttt{chSeller}(s^-)[\\ \quad \mu X.\ s^+?(y)\textbf{. if } \texttt{ok}(y) \textbf{ throw else } X, & \quad \mu X.\ s^-!\langle \texttt{quote} \rangle\textbf{.}\ X\ , \\ \quad s^+!\langle \texttt{card} \rangle\textbf{.}\ s^+?(z)\ ] & \quad s^-?(y_2)\textbf{.}\ s^-!\langle \texttt{time} \rangle\ ] \end{array}$$

Buyer keeps on reading messages on s^+ until condition $\texttt{ok}(y)$ is met and then it throws an exception. Seller, instead, is in an infinite loop where it persistently sends a quote over channel s^- (we assume `quote` changes over time). When the exception is raised the handlers are run: Buyer will send the credit card details `card` and Seller will acknowledge on channel s^- with the current time.

Example 2 (Nested Escapes). The second example given in the introduction, can be represented in our calculus as (Seller is unchanged from Example 1):

$$\begin{array}{ll} \texttt{Buyer} = \overline{\texttt{chBroker}}(t^+)[\ t^+, & \texttt{Broker} = *\texttt{chBroker}(t^-)[\ t^-, \\ \quad t^+!\langle \texttt{id} \rangle\textbf{.} & \quad t^-?(x)\textbf{. if } \texttt{bad}(x) \textbf{ then throw else} \\ & \quad \overline{\texttt{chSeller}}(s^+)[\ (s^+, t^-), \\ \quad \mu X.\ t^+?(y)\textbf{. if } \texttt{ok}(y) \textbf{ throw else } X, & \quad\quad \mu X.\ s^+?(x)\textbf{.}\ t^-!\langle x + 10\% \rangle\textbf{.}\ X, \\ & \quad\quad t^- \triangleleft l_1\textbf{.}\ t^-?(y_2)\textbf{.}\ s^+!\langle y_2 \rangle\textbf{.} \\ \quad t^+ \triangleright \{\ l_1 : t^+!\langle \texttt{card} \rangle\textbf{.}\ t^+?(z), & \quad\quad s^+?(y_3)\textbf{.}\ t^-!\langle y_3 \rangle \quad], \\ \quad\quad l_2 : P_{\texttt{abort}}\} \quad] & \quad t^- \triangleleft l_2\textbf{.}\ R_{\texttt{abort}} \quad] \end{array}$$

Buyer first sends its identity `id` and then Broker throws an exception or proceeds by invoking Seller based on `bad(id)`. In the first case, process $t^- \triangleleft l_2\textbf{.}\ R_{\texttt{abort}}$ in the outermost handler selects the l_2 branch on Buyer's handler and proceeds with abortion (conversation between $P_{\texttt{abort}}$ and $R_{\texttt{abort}}$). In the other case, Seller is invoked and the protocol proceeds as in Example 1 with Broker forwarding messages and increasing quotes by 10%. When Buyer decides to accept a quote, the innermost handler is run by Broker which selects the l_1 conversation in Buyer's handler and forwards the exception to Seller. Then Broker forwards messages, successfully completing the transaction.

Semantics. We augment the semantics of asynchronous sessions [4,9,12] with exception handling (i.e. shutting down a default process and launching the corresponding handler) and exception propagation (informing session peers of an exception occurrence, realised by propagation of the special *exception message* †). Further we ensure that processes always carry out their conversation at properly matching levels (for example when a default process sends a message, a receiving peer may throw an exception before the message arrives, making it no longer relevant), by annotating message queues, hence in effect messages in them, with exception levels.

We use the following *runtime processes* [4,9,12] to define the operational semantics, extending the grammar of programs.

$$P ::= \ldots \mid (\nu s)\, P \quad \text{(resSess)} \qquad \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L \quad \text{(queue)}$$
$$\mid \mathbf{try}\{P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\} \quad \text{(try-catch)} \qquad \mid \tilde{\kappa}[\![P]\!] \quad \text{(wrap)}$$
$$L ::= \epsilon \mid h :: L \qquad h ::= l \mid a \mid \mathsf{tt} \mid \mathsf{ff} \mid \dagger$$

Free variables and channels are extended to run-time processes. Session restriction $(\nu s)\, P$ is standard. For formalising order-preserving asynchronous message passing, we use a directed message queue $\kappa \hookrightarrow_\phi \overline{\kappa} : L$ [4,9], where κ (source) and $\overline{\kappa}$ (target) are two dual endpoint session channels. ϕ ranges over natural numbers, describing the level of the exception at which messages in the queue are to be received, relative to the current position of the queue (we do not need to consider the level of a sender, since this level is recorded by the number of the exception messages $\dagger$ inside a queue). We often write $\kappa \hookrightarrow \overline{\kappa} : L$ for $\kappa \hookrightarrow_0 \overline{\kappa} : L$. The list $L :: h$ is obtained by extending L with an extra tail element h. The *try-catch block* $\mathbf{try}\{P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\}$ is the runtime presentation of a default process and a handler: the default process P in the *try-block* is running during which an exception on channels $\tilde{\kappa}$ can be thrown, which terminates P and launches the handler Q in the *catch-block*. When this Q is launched, it becomes a *wrapped process* or a *wrap*, $\tilde{\kappa}[\![Q]\!]$, making Q immune to an exception notification at the same or upper levels (note such notifications can come due to asynchrony). The transition from a try-catch to a wrap is realised by the meta reduction.

Meta Reduction. The *meta reduction* (1) erases the remaining activity of the default process in the try-block; (2) propagates exceptions to the try-catch blocks inside the try-block; and (3) leaves wrapped processes as they are. In traditional structured exceptions as found in Java or C++, an exception completely erases the try-block and lets the handler run in the same state. In our calculus, concurrently running threads inside a try-block may have conversations (sessions) with other agents. Erasing them would make conversations inconsistent, thus an exception is thrown in each of them.

The meta reduction is written $P \searrow (P', S)$, where the initial process P is transformed into process P', the result of erasing and wrapping; and S denotes session channels via which we should communicate that the exception takes place including the ones of nested try-catch blocks. The rules are defined as follows

$$\text{(MTry)}\quad P \searrow (P', S) \;\Rightarrow\; \mathbf{try}\{P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\} \searrow \begin{cases} (P', S) & \text{if } \tilde{\kappa} \subseteq S \\ (\tilde{\kappa}[\![Q]\!] \mid P', S \cup \tilde{\kappa}) & \text{otherwise} \end{cases}$$
$$\text{(MWrap)}\quad \tilde{\kappa}[\![Q]\!] \searrow (\tilde{\kappa}[\![Q]\!], \emptyset)$$
$$\text{(MPar)}\quad P \searrow (P', S_1) \text{ and } Q \searrow (Q', S_2) \;\Rightarrow\; P \mid Q \searrow (P' \mid Q', S_1 \cup S_2)$$
$$\text{(MNil)}\quad R \searrow (\mathbf{0}, \emptyset) \quad \text{if } R \in \left\{ \begin{array}{l} \text{(inact),(request), (input), (output), (branch),} \\ \text{(select), (cond), (recursion), (throw)} \end{array} \right\}$$

(MTry) propagates the exception to a nested try-catch block. If the try-block meta reduces to some P' with some set S then $\mathbf{try}\{P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\}$ will reduce either to (i) P' itself or to (ii) the parallel composition of P' and $\tilde{\kappa}[\![Q]\!]$ with the new set $S \cup \tilde{\kappa}$ ensuring that also channels $\tilde{\kappa}$ will be notified with an exception. Case (i) discards handler Q when another handler for $\tilde{\kappa}$ is already in P while case (ii) happens when there is no refinement

Table 1. Reduction Semantics

(Init)	$*a(s^-)[P,Q] \mid C[\overline{a}(s^+)[\tilde{\kappa}, P', Q']] \longrightarrow *a(s^-)[P,Q] \mid (\nu s) \begin{pmatrix} \mathbf{try}\{P\}\,\mathbf{catch}\,\{s^- : Q\} & \mid & s^- \hookrightarrow_0 s^+ : \epsilon \mid \\ C[\mathbf{try}\{P'\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q'\}] & \mid & s^+ \hookrightarrow_0 s^- : \epsilon \end{pmatrix}$
(Out)	$\kappa!\langle e\rangle.\, P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : (v :: L) \quad (e \downarrow v)$
(In)	$\kappa?(x).\, P \mid \overline{\kappa} \hookrightarrow_0 \kappa : (L :: v) \longrightarrow P\{v/x\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : L$
(Sel)	$\kappa \triangleleft l.\, P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : (l :: L)$
(Bra)	$\kappa \triangleright \{l_i : P_i\}_{i\in I} \mid \overline{\kappa} \hookrightarrow_0 \kappa : (L :: l_j) \longrightarrow P_j \mid \overline{\kappa} \hookrightarrow_0 \kappa : L \quad (j \in I)$
(Con)	$P \longrightarrow Q \Rightarrow C[P] \longrightarrow C[Q]$
(If)	$\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \longrightarrow P \quad (e \downarrow \mathsf{tt})$ $\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \longrightarrow Q \quad (e \downarrow \mathsf{ff})$
(Str)	$P \equiv P'$ and $P' \longrightarrow Q'$ and $Q' \equiv Q \Rightarrow P \longrightarrow Q$
(Thr)	$\mathbf{try}\{P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\} \searrow (R,S) \Rightarrow \mathbf{try}\{\mathbf{throw} \mid P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\} \mid \Pi_{\kappa\in S}\, \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow R \mid \Pi_{\kappa\in S}\, \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$
(RThr)	$\mathbf{try}\{P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\} \searrow (R,S) \Rightarrow \mathbf{try}\{P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\} \mid \overline{\kappa}_j \hookrightarrow_0 \kappa_j : (L :: \dagger) \mid \Pi_{\kappa\in S}\, \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow R \mid \overline{\kappa}_j \hookrightarrow_1 \kappa_j : L \mid \Pi_{\kappa\in S}\, \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$
(WVal)	$\tilde{\kappa}\{\!|Q|\!\} \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: v) \longrightarrow \tilde{\kappa}\{\!|Q|\!\} \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : L$
(WThr)	$\tilde{\kappa}\{\!|Q|\!\} \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: \dagger) \longrightarrow \tilde{\kappa}\{\!|Q|\!\} \mid \overline{\kappa}_i \hookrightarrow_1 \kappa_i : L$
(Clean)	$P \searrow (R,S),\ (\lambda \in \tilde{\kappa}, \dagger \in L) \Rightarrow \mathbf{try}\{P \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L\}\,\mathbf{catch}\,\{\tilde{\kappa} : \tilde{Q}\} \mid \Pi_{\kappa\in S}\, \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow R \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L \mid \Pi_{\kappa\in S}\, \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$

of $\tilde{\kappa}$ in P. The mechanism is sound because of the assumption that κ_i are always refined together (cf. syntax). Note that, if the try-block is single-threaded, the meta reduction mechanism is identical to the one of standard exception handling.

Reduction. We now introduce the main reduction rules. Due to the nesting of wraps and try-catch blocks, the reduction is defined using the following reduction context:

$$C \ ::= \mathbf{try}\{C\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\} \mid P \mid C \mid \tilde{\kappa}\{\!|C|\!\} \mid (\nu s)\, C \mid (\nu a)\, C \mid -$$

The reduction $\longrightarrow$ is the smallest relation generated by the rules in Table 1. (Init) gives the semantics of session initiation, generating two fresh dual session channels, the associated two empty queues (ϵ denotes the empty string) and the two try-catch blocks $\mathbf{try}\{P\}\,\mathbf{catch}\,\{s^- : Q\}$ and $\mathbf{try}\{P'\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q'\}$. Note that $*a(s^-)[P,Q]$ is not in a context. This is because we have assumed that *services never appear nested in a try- or a catch-block* as we do not want them to be terminated (following SCP).

(Out) and (Sel) enqueue, respectively, a value and a label at the head of the queue for κ. Symmetrically, (In) and (Bra) dequeue from the tail of the queue. The exception level in the latter two rules is 0, indicating the level of an actual receiver. The exception level of a queue ensures that a message is sent and received at the same level, guaranteeing

consistency of communication. This depends on the invariance that the sum of the level of the queue and the number of †'s in the queue before a specific message, determines the depth (the number of wraps) at which the message enqueueing is performed. In (Out,If), $e \downarrow v$ says that expression e evaluates to value v. (Con,Str) are standard.

(Thr) and (RThr) represent the firing of an exception. (Thr) is when **throw** appears top-level in the try-block, i.e. exception is thrown locally; while (RThr) is when a remote exception is received as † in the queue. Eventually, all peers will be notified of the exception by sending † via channels in S generated from P as well as $\tilde{\kappa}$. An alternative semantics prioritises † [2].

Rule (WVal) describes the case when messages at the default level meet a wrapped process and are drained into a sink (i.e. get dequeued but ignored). In (WThr), † meets a wrap and the exception level of the queue is incremented, allowing the queue to enter the wrap. In (Clean), † in the queue reveals the presence of a refinement in P which has now become a wrap due to a local throw. Meta reduction propagates the exception to each parallel process in P and the try-catch block is discarded.

This last step is formally defined by the structural congruence $\equiv$ which plays a key role in treating exceptions and, in particular, moving queues while maintaining their exception levels. $\equiv$ is the least congruence relation on processes such that $(P, \mid)$ is a commutative monoid and includes the standard rules for restriction (such as scope extrusion) and recursion. In addition, it has the following rules:

a) $\mathbf{try}\{P \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L\}\ \mathbf{catch}\ \{\tilde{\kappa} : Q\} \equiv \mathbf{try}\{P\}\ \mathbf{catch}\ \{\tilde{\kappa} : Q\} \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L \quad (\lambda \in \tilde{\kappa} \Rightarrow \dagger \notin L)$

b) $\tilde{\kappa}\{\!|P \mid \overline{\lambda} \hookrightarrow_\phi \lambda : L|\!\} \equiv \tilde{\kappa}\{\!|P|\!\} \mid \overline{\lambda} \hookrightarrow_\phi \lambda : L \qquad (\lambda \notin \tilde{\kappa})$

c) $\tilde{\kappa}\{\!|P|\!\} \mid \overline{\kappa}_i \hookrightarrow_\phi \kappa_i : L \equiv \tilde{\kappa}\{\!|P \mid \overline{\kappa}_i \hookrightarrow_{\phi-1} \kappa_i : L|\!\}$

d) $\mathbf{try}\{(\nu a)\, P\}\ \mathbf{catch}\ \{\tilde{\kappa} : Q\} \equiv (\nu a)\, \mathbf{try}\{P\}\ \mathbf{catch}\ \{\tilde{\kappa} : Q\} \qquad (a \notin \mathsf{fn}(Q))$

e) $\tilde{\kappa}\{\!|(\nu a)\, P|\!\} \equiv (\nu a)\, \tilde{\kappa}\{\!|P|\!\}$

The first and second rules allow a queue to move into a try-catch block and a wrap respectively. The third rule is applicable when the receiving side of the queue is in $\tilde{\kappa}$: when entering the wrap, ϕ is decreased so that the process inside the wrap can read the value if the level after the decrement is 0. The last two rules open the scope.

To illustrate how queue levels work, we consider the following process:

$$P = \mathbf{try}\{\ \mathbf{throw} \mid \kappa!\langle 5\rangle\ \}\ \mathbf{catch}\ \{\kappa : \kappa!\langle \mathtt{tt}\rangle\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : \epsilon \mid \\ \mathbf{try}\{\ \mathbf{throw} \mid \overline{\kappa}?(x)\ \}\ \mathbf{catch}\ \{\overline{\kappa} : \overline{\kappa}?(x)\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon$$

Process P can reduce to $P' = \kappa\{\!|\mathbf{0}|\!\} \mid \overline{\kappa}\{\!|\mathbf{0}|\!\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : \epsilon \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon$ in different ways.

$$\begin{aligned} P &\longrightarrow\equiv \kappa\{\!|\kappa!\langle \mathtt{tt}\rangle|\!\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger \mid \mathbf{try}\{\ \mathbf{throw} \mid \overline{\kappa}?(x)\ \}\ \mathbf{catch}\ \{\overline{\kappa} : \overline{\kappa}?(x)\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon \\ &\longrightarrow\equiv \kappa\{\!|\mathbf{0}|\!\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : (\mathtt{tt} :: \dagger) \mid \mathbf{try}\{\ \mathbf{throw} \mid \overline{\kappa}?(x)\ \}\ \mathbf{catch}\ \{\overline{\kappa} : \overline{\kappa}?(x)\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon \\ &\longrightarrow\equiv \kappa\{\!|\mathbf{0}|\!\} \mid \kappa \hookrightarrow_1 \overline{\kappa} : \mathtt{tt} \mid \overline{\kappa}\{\!|\overline{\kappa}?(x)|\!\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \\ &\longrightarrow\equiv \kappa\{\!|\mathbf{0}|\!\} \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon \mid \kappa \hookrightarrow_1 \overline{\kappa} : \mathtt{tt} \mid \overline{\kappa}\{\!|\overline{\kappa}?(x)|\!\} \longrightarrow\equiv P' \end{aligned}$$

In this case, an exception and then `tt` are sent over κ. Finally the exception is delivered to $\overline{\kappa}$ before delivering `tt`. But we can also have:

$$\begin{aligned} P &\longrightarrow\longrightarrow\equiv \mathbf{try}\{\ \mathbf{throw}\ \}\ \mathbf{catch}\ \{\kappa : \kappa!\langle \mathtt{tt}\rangle\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : 5 \mid \overline{\kappa}\{\!|\overline{\kappa}?(x)|\!\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \\ &\longrightarrow\equiv \kappa\{\!|\kappa!\langle \mathtt{tt}\rangle|\!\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : (\dagger :: 5) \mid \overline{\kappa}\{\!|\overline{\kappa}?(x)|\!\} \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon \longrightarrow\longrightarrow\longrightarrow\equiv P' \end{aligned}$$

Above, 5 is sent over κ and an exception is thrown on $\overline{\kappa}$. In this situation, the system will ignore 5 (discarded by (WVal)), and deliver `tt` inside the wrap.

The following example shows how refinement of an existing exception is handled:

$$R \;=\; \textbf{try}\{\ \textbf{try}\{\ \textbf{throw}\ \}\ \textbf{catch}\ \{\,(\kappa,\lambda) : Q_1\,\}\ \}\ \textbf{catch}\ \{\,\kappa : Q_2\,\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \mid \kappa \hookrightarrow_0 \overline{\kappa} : L$$

Process R either throws an exception in the inner try-catch block (by (Thr)) or receives a remote exception (by (RThr)). By applying (Thr), (Clean) and (WThr) in the first case or by (RThr) in the second case, we have (omitting some queues):

$$R \longrightarrow\equiv\ \textbf{try}\{\ (\kappa,\lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L\,\}\ \textbf{catch}\ \{\,\kappa : Q_2\,\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \longrightarrow$$
$$(\kappa,\lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \longrightarrow (\kappa,\lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon.$$

3 Typing Interactional Exceptions

This section introduces a type discipline for sessions with interactional exceptions. In comparison with the standard session types, the central difference is the shape of a type itself, which now consists of the abstraction of the default behaviour (the "try" part) and that of the handler behaviour (the "catch" part). This simple extension, combined with the use of levels, allows to establish subject reduction, guaranteeing that messages are always delivered at proper levels at proper timings in the presence of nested asynchronous escapes, testifying consistency of the operational semantics introduced in §2.

Type Syntax. The grammar of types extends the standard session types:

$$\begin{array}{rcl} \alpha,\beta & ::= & \downarrow(\theta).\,\alpha \mid \uparrow(\theta).\,\alpha \mid \oplus\{l_i : \alpha_i\}_{i\in I} \mid \&\{l_i : \alpha_i\}_{i\in I} \mid \alpha[\![\beta]\!] \mid \mathtt{end} \mid \mu\mathbf{t}.\,\alpha \mid \mathbf{t} \\ \theta & ::= & \langle\alpha[\![\beta]\!]\rangle \mid \mathtt{bool} \mid \ldots \end{array}$$

α and θ are respectively called *session types* and *service types*. The grammar follows the standard session types [10, 17], except for *try-catch type* $\alpha[\![\beta]\!]$, the abstraction of a try-catch block: in $\alpha[\![\beta]\!]$, α denotes the type of the try-block and β the catch block. A session type α is *plain* if it does not use a try-catch type (except in a service type it carries). From now on in $\alpha[\![\beta]\!]$, we stipulate α and β are both plain. This is because a try-catch on κ cannot occur nested in a try- or catch-block of λ if $\kappa = \lambda$.

The *dual* of α is written $\overline{\alpha}$. The dual of the try-catch type is defined as $\overline{\alpha[\![\beta]\!]} = \overline{\alpha}[\![\overline{\beta}]\!]$: the other cases are standard [10]. For example, by exchanging input and output, the dual of $\downarrow(\mathtt{string}).\mathtt{end}[\![\uparrow(\mathtt{bool}).\mathtt{end}]\!]$ is $\uparrow(\mathtt{string}).\mathtt{end}[\![\downarrow(\mathtt{bool}).\mathtt{end}]\!]$.

Environments. *Typing judgements* for processes and expressions have the forms $\Gamma \vdash P \triangleright \Delta$ and $\Gamma \vdash e : \theta$ respectively where Γ is a *service typing*, which typically maps service channels to service types and Δ is a *session typing* which typically maps session channels to session types. For ($n \in \{0,1\}$ and $\rho \in \{p,u\}$), typings are defined as

$$\begin{array}{lll} \textit{(Session Typing)} & \Delta & ::= \emptyset \mid \Delta,\ \kappa :^n_\rho \alpha \mid \Delta,\ (\kappa,\overline{\kappa}) : \alpha \mid \Delta,\ (\kappa,\overline{\kappa}) : \bot \\ \textit{(Service Typing)} & \Gamma & ::= \emptyset \mid \Gamma,\ c : \langle\alpha[\![\beta]\!]\rangle \mid c : \mathtt{bool} \mid \Gamma,\ X : \Delta \end{array}$$

In session typings, $\kappa :^n_\rho \alpha$ says that: *at a polarised session channel κ, there is a session of type α*. The natural number n is equal to 1 if there is a wrap on κ, 0 otherwise. A session channel with respect to its type is *unprotected* if $\rho = u$ (no try-catch nor wrap on κ occurs) and *protected* if $\rho = p$ (there is a try-catch or a wrap on κ). This is needed in the try-catch and wrap typing as well as in the merging with the queue types $(\kappa,\overline{\kappa}) : \alpha$

and $(\kappa, \overline{\kappa}) : \bot$ used for typing a queue from κ to $\overline{\kappa}$ (the type of a queue is composed with the type of a process in which case the queue's type becomes $\bot$).

In the service typing, c either has type $\alpha\{\!|\beta|\!\}$ (a service using a session channel with default behaviour of type α and with a handler of type β) or an atomic type such as `bool`. Typing $X : \Delta$ is used for recursion as in [6].

Typing System for Programs. We show the typing system by which the programmer can check whether her program is error free or not, especially w.r.t. its exception usage. The following are the selected typing rules:

$$(\text{TReq})\ \frac{\begin{array}{l}\Gamma \vdash P \triangleright \prod_i \kappa_i :^0_u \overline{\alpha}_i\{\!|\overline{\beta}_i|\!\} \\ \Gamma' \vdash Q \triangleright \prod_i \kappa_i :^0_u \overline{\beta}_i \qquad s^+ = \kappa_j \\ \Gamma \vdash c : \langle \alpha_j\{\!|\beta_j|\!\}\rangle \qquad \Gamma' \subseteq \Gamma,\ \mathsf{fv}(\Gamma') = \emptyset\end{array}}{\Gamma \vdash \overline{c}(s^+)[\tilde{\kappa}, P, Q] \triangleright \prod_{i \neq j} \kappa_i :^0_u \overline{\alpha}_i\{\!|\overline{\beta}_i|\!\}} \qquad (\text{TServ})\ \frac{\begin{array}{l}\Gamma \vdash P \triangleright s^- :^0_u \alpha\{\!|\beta|\!\} \\ \Gamma \vdash Q \triangleright s^- :^0_u \beta \qquad \mathsf{fv}(\Gamma) = \emptyset\end{array}}{\Gamma,\ a : \langle \alpha\{\!|\beta|\!\}\rangle \vdash *a(s^-)[P, Q] \triangleright \emptyset}$$

$$(\text{TThr})\ \frac{\mathsf{fv}(\Gamma) = \emptyset}{\Gamma \vdash \mathbf{throw} \triangleright \prod_i \kappa_i :^0_u \alpha_i} \qquad (\text{TPar})\ \frac{\Gamma \vdash P_i \triangleright \Delta_i\ (i = 1, 2)\ \ \Delta_1 \asymp \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \odot \Delta_2}$$

Other than the rules for the exception constructs, all rules are identical to [21], augmented with annotation of exception levels.

(TReq) types a request on service channel c whose type, according to Γ, is $\alpha_j\{\!|\beta_j|\!\}$. Condition $s^+ = \kappa_j$ makes sure that the fresh name s^+ will also be in the try-catch after reduction. Session s^+ has type $\overline{\alpha}_j\{\!|\overline{\beta}_j|\!\}$, the dual of c's type. This rule checks that each κ_i in Q (exception handler) has type $\overline{\beta}_i$ whereas in P it has type $\overline{\alpha}_i\{\!|\overline{\beta}_i|\!\}$ where each $\overline{\beta}_i$ may come from a refinement of κ_i in P. Finally, Γ' is a subset of Γ without free variables for service channels (otherwise the queue stores open terms at run-time). In (TServ), because of SCP in § 2, services should never be prefixed therefore the only visible (free) session in P and Q should be s^-. Throwing an exception interrupts any conversation, thus (TThr) allows to type **throw** with any $\kappa : \alpha$ (unprotected). (TPar) requires the coherence relation $\asymp$ and the partial operator $\odot$ based on duality [10, 17]. When typing programs, the operator becomes just a set union. We shall extend it for types of queues in the next subsection. The rules for communication are standard.

Typing System for Run-Time Processes. The rules for typing run-time processes, which are necessary for type soundness, include, among others:

$$(\text{TExcept})\ \frac{\begin{array}{l}\Gamma \vdash P \triangleright \Delta \cdot \prod_j \lambda_j :^{n_j}_p \alpha'_j \cdot \prod_i \kappa_i :^{m_i}_\rho \alpha_i\{\!|\beta_i|\!\} \\ \Gamma' \vdash Q \triangleright \prod_i \kappa_i :^0_u \beta_i \qquad \mathsf{queue}(\Delta) \qquad \Gamma' \subseteq \Gamma,\ \mathsf{fv}(\Gamma') = \emptyset\end{array}}{\Gamma \vdash \mathbf{try}\{P\}\,\mathbf{catch}\,\{\tilde{\kappa} : Q\} \triangleright \Delta \cdot \prod_j \lambda_j :^{n_j}_p \alpha'_j \cdot \prod_i \kappa_i :^{m_i}_p \alpha_i\{\!|\beta_i|\!\}}$$

$$(\text{TWrap})\ \frac{\Gamma \vdash Q \triangleright \Delta \cdot \prod_i \lambda_i :^{n_i}_p \alpha'_i \cdot \prod_i \kappa_i :^0_u \beta_i \qquad \mathsf{queue}(\Delta)}{\Gamma \vdash \tilde{\kappa}\{\!|Q|\!\} \triangleright \Delta \cdot \prod_i \lambda_i :^{n_i}_p \alpha'_i \cdot \prod_i \kappa_i :^1_p \alpha_i\{\!|\beta_i|\!\}}$$

(TExcept) is a key rule, associating the type $\alpha\{\!|\beta|\!\}$ to the try-catch block, ensuring each κ_i is used as $\alpha_i\{\!|\beta_i|\!\}$ in P (β_i may come from a refinement in P) and as β_i in Q. The premise makes sure that each $\lambda_j \notin \tilde{\kappa}$ is protected in a try-catch block or a wrap in P. Without this condition, we may end up with unprotected code that could be brutally removed by an exception, violating the session duality. For example, in $\mathbf{try}\{\lambda!\langle v\rangle.\ R\}\,\mathbf{catch}\,\{\kappa : Q\} \mid \overline{\lambda}?(x).\ R \mid P$, if an exception is thrown by P over κ,

the output $\lambda!\langle v\rangle$**.** R would be lost leaving the input $\overline{\lambda}?(x)$**.** R alone violating duality. The predicate $\mathsf{queue}(\Delta)$ checks that Δ contains queue types only. No condition on m_i is required since in P, each κ_i can be either unprotected (no wraps nor try-catch blocks on κ_i) or protected (because of a refinement, κ_i occurs in a try-catch block or in a wrap). Finally, in order to record that now we have the try-catch block on κ_i, we force the tag ρ to become p, i.e. protected. Note that, since Q is a program, we do not need to check that $\mathsf{fv}(\Gamma) = \emptyset$.

(TWRAP) types a wrap over a process Q. All the κ_i that have type β_i will have new type $\alpha_i[\![\beta_i]\!]$ so to form the correct dual for the other side of the session in the case the exception has not been yet received there. The new type will make sure that each κ_i is protected. We set n to 1 in order to remember an existence of a wrap. As queues contain only an output or a select, queue types will only have output or selection types [12].

We conclude with the definition of $\asymp$ and $\odot$ using the treatment of queue types from [12]. We say Δ_1 and Δ_2 are *compatible*, written $\Delta_1 \asymp \Delta_2$, if and only if (i) $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) = \emptyset$; (ii) $\kappa :^n_\rho \alpha$, $(\kappa, \overline{\kappa}) :\perp$, $\overline{\kappa} :^m_{\rho'} \beta$, $(\overline{\kappa}, \kappa) :\perp \in \Delta_1 \cup \Delta_2$ implies $\alpha = \overline{\beta}$; (iii) $\kappa :^n_\rho \alpha$, $(\kappa, \overline{\kappa}) : \alpha'$, $\overline{\kappa} :^m_{\rho'} \beta$, $(\overline{\kappa}, \kappa) : \beta' \in \Delta_1 \cup \Delta_2$ implies $\mathsf{merge}(\alpha', \alpha, n) = \overline{\mathsf{merge}(\beta', \beta, m)}$; and (iv) $\kappa :^n_\rho \alpha$, $(\kappa, \overline{\kappa}) :\perp$, $\overline{\kappa} :^m_{\rho'} \beta$, $(\overline{\kappa}, \kappa) : \beta' \in \Delta_1 \cup \Delta_2$ implies $\alpha = \overline{\mathsf{merge}(\beta', \beta, m)}$. The operation $\mathsf{merge}(\alpha', \alpha[\![\beta]\!], n)$ merges a session type with the type of the queue w.r.t. the level n i.e. it merges α' with α if $n = 0$ and it merges α' with β if $n = 1$. The operation $\Delta_1 \odot \Delta_2$ (defined if $\Delta_1 \asymp \Delta_2$) is such that if $\kappa :^n_\rho \beta$ and $(\kappa, \overline{\kappa}) : \alpha$ are in $\Delta_1 \cup \Delta_2$ then $\kappa :^n_\rho \mathsf{merge}(\alpha, \beta, n), (\kappa, \overline{\kappa}) :\perp \in \Delta_1 \odot \Delta_2$ ($\perp$ keeps track that the corresponding queue exists). The other elements in $\Delta_1 \odot \Delta_2$ are the same as in $\Delta_1 \cup \Delta_2$.

The processes in Examples 1/2 are typable: channel chSeller in both examples has type $\mu\mathbf{t}.\ \uparrow(\mathtt{int}).\ \mathbf{t}[\![\downarrow(\mathtt{int}).\ \uparrow(\mathtt{time})]\!]$. In Example 2, channel chBroker has type $(\downarrow(\mathtt{int}).\ \mu\mathbf{t}.\uparrow(\mathtt{int}).\ \mathbf{t})[\![\oplus\{\mathtt{l}_1 :\downarrow(\mathtt{int}).\ \uparrow(\mathtt{time}), \mathtt{l}_2 : \alpha\}]\!]$ for some α.

In the following Theorem, $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$.

Theorem 1 (Subject Reduction). *Let P be a program such that $\Gamma \vdash P \triangleright \emptyset$. If $P \longrightarrow^* Q$ then $\Gamma \vdash Q \triangleright \emptyset$.*

As a corollary, the typing system also satisfies type safety and communication safety including communication-error freedom and linearity [12, Theorem 5.5].

4 Liveness

The previous section establishes a basic consistency of dynamics of typed processes via subject reduction. This section strengthens this result with a liveness property, which intuitively says that all compensating session channels eventually get fired (except those which are "erased" by thrown exceptions in which case the compensating actions both disappear). Along the way we also show relative termination of individual protocols for exceptions and their partial confluence vis-a-vis ordinary communications, two key consistency criteria for the proposed operational mechanism.

A Termination Protocol. We first augment the operational semantics in §2 with yet another protocol, called *termination protocol*, which in fact is an intrinsic part of the dynamics of exceptions. As an example, suppose there are two (and only two) processes in

a configuration, which are try-catch blocks and which are communicating in a session. If each party's default process becomes the inaction process, it is natural to reduce each try-catch block to the inaction, freeing up the resources for its handler. This "garbage collection" is essential when we consider integration of interactional exception into the standard imperative programming languages with sequential composition since in this case launching Q depends on whether P reduces to the inaction or not.

We call a termination of a try-block in this form, *normal exit*. In interactional exceptions, a normal exit of a try-catch block demands an agreement of all peers: even if one try-block has terminated, if any of its communicating peers throws an exception, it also has to throw one, hence synchronising among all peers is essential for consistency. The termination protocol we introduce below makes the most of the tree structure associated with hierarchy of service invocation, leading to relatively efficient execution in terms of the number of messages. The protocol consists of two stages:

(**Stage 1: Voting):** Each try-catch block notifies its caller in the caller-callee relation of services by sending its vote of termination after itself terminating and receiving the same news from its callees.
(**Stage 2: Decision):** When the initial caller hears this, it in turn lets the news flow down to all of its callees, upon whose receipt the try-catch blocks can normally terminate.

Above, callees and callers refer to the service invocation mechanism. We formalise this protocol by extending the reduction relation. First, we augment polarities of channels in try-catch blocks with $\{\oplus, \ominus\}$, replacing $+, -$ for Stage 2 (indicating the casting of votes). We also use two special messages, $\{V, D\}$ standing for Voting and Decision.

$$
\begin{array}{ll}
(\textsc{Coll}) & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s'^{-}; \tilde{t}^{\oplus}; s^{+}; \tilde{r}^{+} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : V \quad \to \\
 & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s'^{-}; \tilde{t}^{\oplus}; s^{\oplus}; \tilde{r}^{+} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : \epsilon \\
(\textsc{Vote}) & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s^{-}; \tilde{t}^{\oplus} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : \epsilon \to \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s^{\ominus}; \tilde{t}^{\oplus} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : V \\
(\textsc{Root}) & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{\tilde{s}^{\oplus} : \tilde{Q}\} \mid \textstyle\prod_i s_i^{+} \hookrightarrow_{\phi} s_i^{-} : \epsilon \;\to\; \star \mid \prod_i s_i^{+} \hookrightarrow_{\phi} s_i^{-} : D \\
(\textsc{Dec}) & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s^{\ominus}; \tilde{t}^{\oplus} : \tilde{Q}\} \mid s^{+} \hookrightarrow_{\phi} s^{-} : D \mid \textstyle\prod_i t_i^{+} \hookrightarrow_{\phi} t_i^{-} : \epsilon \quad \to \\
 & \star \mid s^{+} \hookrightarrow_{\phi} s^{-} : \epsilon \mid \textstyle\prod_i t_i^{+} \hookrightarrow_{\phi} t_i^{-} : D
\end{array}
$$

where $\star ::= \mathbf{0} \mid \star \mid \star \mid \tilde{x}[\![\star]\!]$. Briefly, (Coll) collects the votes from the callees; (Vote) sends a vote to the caller once all the callees have voted; (Root) is for the initial caller which terminates once all its callees have voted; and (Dec) terminates once the caller has terminated. The rules only make sense for well-typed processes as we shall show later. We write $P \to_{term} P'$ for a reduction generated from the above rules. As an example, consider the following processes:

$$\mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{s^{-}; t^{+}; r^{+} : Q_1\} \mid \mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{s^{+} : Q_2\} \tag{1}$$

$$\mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{t^{-} : Q_3\} \mid \mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{r^{-} : Q_4\} \tag{2}$$

By (Vote) the two processes in (2) will put V in the queues with writing side r^{+} and t^{+} respectively. Applying (Coll) twice, the right-hand process in (1) will reach the state $\mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{s^{-}; t^{\oplus}; r^{\oplus} : Q_2\}$. Again, by (Vote), it will send V to the parent. Finally, by (Root) and then (Dec) it will reduce to process $\mathbf{0}$.

Note that for the exit protocol to work, the caller-callee relation must have a tree structure where the root is the initial service invoker and leaves are a collection of interacting processes. We shall explore this topic further in the next subsection.

Normal and Exceptional Exits. We first show that the above protocol always leads to a normal exit. For this purpose, we need to identify the set of nodes that form the caller-callee relation in a given process.

Definition 2 (Node). *R is a* node process *of P whenever $P \equiv (\nu \tilde{s})\,(Q \mid R)$, R is restriction/queue/service-free, and R does not have the form $\star$ or $R' \mid R''$. If $\mathsf{fsc}(R) \neq \emptyset$ then $\mathsf{fsc}(R)$ is called* a node *of P.*

A node process of P is a subprocess of P which is not the parallel composition of two other processes and contains no restriction, queue, service or is composed by zero or more wraps over process $\mathbf{0}$. Then a node is the set of free session channels (only if non-empty) of a node process. Using the processes (1) and (2) given above, the process (1) | (2) has four nodes: s^+, (s^-, t^+, r^+), t^- and r^-. Process $\mathbf{try}\{\kappa!\langle v\rangle.\ \mathbf{0}\}\ \mathbf{catch}\ \{\kappa : Q\}$ has a unique node κ while $\mathbf{try}\{\kappa!\langle v\rangle.\ \mathbf{0}\}\ \mathbf{catch}\ \{\kappa : Q\} \mid \lambda?(x)$ also has λ. The caller-callee structure of a process is identified by the directed edges of the following graph.

Definition 3 (Invocation Graph). *Let G be the set of nodes of a process P. Then the* invocation graph of P *is the directed graph $\mathcal{G} = (G, E)$ where $(\tilde{\kappa}, \tilde{\lambda}) \in E$ if and only if $\kappa_i \in \{s^+, s^{\oplus}\}$ and $\lambda_j \in \{s^-, s^{\ominus}\}$ for some s, i, j. An* invocation tree *in P is a maximal subtree in the invocation graph.*

The invocation graph of process (1) | (2) is a graph with one edge from s^+ to (s^-, t^+, r^+), one from (s^-, t^+, r^+) to t^- and one from (s^-, t^+, r^+) to r^-. Recall the definition of programs given in § 2.1. If we reduce a typed program by zero or more steps then the invocation graph of the resulting process always forms a forest.

Lemma 4 (Evolution). *Let P be a typable program. If $P \to^* R$ then R's invocation graph $\mathcal{G}(R)$ is a forest.*

An invocation tree in P is in the *pretermination state* if each of its try-blocks have the form $\star$. From the Evolution Lemma, it follows that once an invocation tree reaches a pretermination state then all of its nodes will eventually vanish (see [2] for details).

Theorem 5 (Normal Exit). *Let P_0 be typable program and $P_0 \to^* P$ such that P has an invocation tree $\mathcal{T}$ in the pretermination state. Then whenever $P \to^* P'$ there is Q such that $P' \to^*_{term} Q$ where Q does not contain any active nodes from $\mathcal{T}$.*

The result above can actually be made stronger. For instance, in $\mathbf{try}\{\mathbf{try}\{\mathbf{0}\}\ \mathbf{catch}\ \{\lambda : Q_1\}\}\ \mathbf{catch}\ \{\kappa : Q_2\}$, we do not have a pretermination state as the outer try-block contains a try-catch block. Nevertheless, the normal exit is guaranteed as when the inner block and the subtree connected to λ terminate, the outer catch block can proceed.

A try-catch block can also exit due to an exception which will be propagated through the invocation graph. We write $P \to_{ex} P'$ if this is generated from (RTHR) and we say that κ is in *preexception* if it is the channel for a try-block which moreover contains an active **throw**.

Theorem 6 (Exception Exit). *Let P_0 be a typable program and $P_0 \to^* P$ such that P has an invocation tree $\mathcal{T}$. Suppose $\tilde{\kappa}$ is in a node of $\mathcal{T}$ which is in preexception for some κ_i. Then $P \to^* P'$ implies $P' \to^*_{ex} R$ for some R.*

Liveness. We can use the Evolution Lemma to obtain a strong form of liveness for well-typed processes in the presence of asynchronous exceptions. We first define:

Definition 7. *We say P is* stable *if P is the parallel composition of zero or more $\star$ processes and zero or more empty queues, possibly under ν-restrictions. We say P has* all resources *if $a \in \mathsf{fn}(P)$ implies $P \equiv (\nu \tilde{u})\,(*a(\lambda)[Q_1, Q_2] \mid R)$, i.e. all output channels are compensated by replicated inputs.*

We can finally state that a well-typed program either continues to reduce forever or reaches a state which does not contain active prefixes (except replicated services), try-catch blocks, throws nor messages in transit.

Theorem 8 (Liveness). *Suppose P is a typable program and has all resources. Then $P \rightarrow^* Q$ implies either Q is stable or $Q \rightarrow Q'$ for some Q'.*

In particular, if there are two compensating actions in a session at the same exception level, then these two will eventually interact, attesting the consistency of exception protocols. Practically, observing that SCP (the key reason the result holds) is widely found in e.g. services in the world wide web, the result says that if a conversation ever gets stuck in such an environment, it may as well be for non-interactional reasons (such as deadlock over shared resources at the servers).

5 Related Work and Conclusion

Related Work. In concurrent programming of distributed objects, exception handling is investigated in [20] where an algorithm to resolve multiple kinds of exceptions (which form a linear order) among concurrently running objects is proposed. Asynchronous exceptions among concurrent threads and their interplay with states in Haskell is studied in [16]. Motivated by subtle race conditions for mutual states, they formalise and implement blocking constructs to postpone asynchronous exceptions. The key idea is to relax the exception mask through the use of interruptible operations, to balance asynchrony and state consistency. [3] introduces a model for long-running transactions which treats failures by restoring the initial state and firing a compensation process. The calculus for web services called COWS [15] provides an operation to kill processes except those protected by wraps similar to our exception mechanism. CaSPiS [5], a session-based process calculus, is equipped with an operator for session closures. Our termination protocol, instead, is run whenever a try-block contains an inactive process, ensuring liveness. [18] introduces a calculus for web services by extending the π-calculus with service and request primitives and local exceptions, without asynchronous queues. An interesting idea is *context*, a named tree-like structure where a process is located. They do not have an explicit notion of session type. Their exceptions are the traditional local exceptions, without supporting propagation, coordinated transfer to a different part of a dialogue, nor the associated type abstraction, so that type checking protocols with exceptions, such as Examples 1/2 in §2, would be difficult.

The central focus of the present work is to have basic high-level typed abstractions for clear and flexible descriptions of conversation structures. Exceptions are asynchronously raised by multiple communicating peers, for which the session compatibility can guarantee type-safety in the presence of arbitrarily nested exceptions. These key aspects, backed up by safety and liveness properties relying on linearity of session-based communications, have not been investigated in the existing studies.

Further Results. For the sake of simplicity, we have restricted programs so that in $*c(\lambda)[P, Q]$ and $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$, the handler Q does not contain another try-catch at the same λ (try-catch is only used at run-time). An extension of our formalism allowing try-catch to occur in the handlers of (service) and (request) would allow a process to "try" again after an exception has been thrown (cascading exceptions). For this purpose, try-catch types should be extended such that in $\alpha\{\!|\beta|\!\}$ the type α is always plain while β can be either plain or a try-catch type. Additionally, as it is now possible to have any number of nested wraps (when an exception is thrown several times), the number n in $\kappa :^{n}_{\rho} \alpha$ becomes arbitrary natural numbers, generalising their composition with queue types (n wraps). With essentially the same operational semantics, this generalised calculus satisfies the subject reduction and liveness properties. Further generalisation to existing session types is possible, including multiparty sessions [12] for flexible multicast exception propagation.

Further Topics. The key idea of the presented operational semantics is the use of exception levels in queues and their interplay with wrapped processes. In implementation, the queue level can be recorded in a header of each message which its receiver can check efficiently. The wrapping level can be part of a process state, recording its exception depth. Various optimisations are possible, for example dispensing with most coordination protocols when the handler type is trivial, obtaining essentially the same level of efficiency as local exception. In the near future, we plan to incorporate this exception mechanism to our on-going implementation of Java with session types [13].

For simplicity, we omit session delegations: we formulated this extension by storing frozen processes in queues. The type soundness holds by extending the typing rules with those in [10]. However a construction of the invocation graphs which can guarantee forest structures for the liveness property is left open.

Our liveness property, which involves the termination protocols, is similar to the property found in e.g. [7]. Apart from presence of exceptions, the aims and approaches of the two works are quite different: in the present work, the liveness is used for ensuring consistency of the proposed exception mechanisms, and the proof method is more operational, being applicable to any session-based calculus which has the service channel principle without delegation, without changing its typing system. On the other hand, in [7], an effect-based typing system is used for progress with delegation. It is an interesting further topic to incorporate our typing system with [7] for obtaining liveness with both exceptions and delegations.

A significant future topic is the treatment of multiple kinds of exception types in the present framework. Following [20], we may assume ordering on exception types (such as the exception class hierarchy as found in Java) for coordinating exceptions among multiple peers. Using a similar technique developed in our termination protocol, we can exploit the tree-structure of an invocation graph for efficient resolution of exception types to handle the subtle interplay between multiple exception types and nested exceptions for a more refined exception propagation.

With interactional exceptions, many practical scenarios can be accurately described through session primitives, and type-checked by our type theory. The syntax and type structures developed in this paper are being considered for use in a Web Services

language (WS-CDL [6,19]) and a language of message schemes for financial communications (ISO 20022 UNIFI [14]), throughout our collaborations with industry partners.

Acknowledgements. We thank the reviewers for their comments and our academic and industry colleagues for their stimulating conversations. This work is partially supported by EPSRC GR/T03208, EP/F002114 and EP/F003757, and IST2005-015905 MOBIUS.

References

1. Advanced Message Queuing Protocol, http://www.iona.com/opensource/amqp/
2. Long version of this paper, http://www.dcs.qmul.ac.uk/~carbonem/exception
3. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 124–138. Springer, Heidelberg (2003)
4. Bonelli, E., Compagnoni, A.: Multipoint Session Types for a Distributed Calculus. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 240–256. Springer, Heidelberg (2008)
5. Boreale, M., Bruni, R., Nicola, R.D., Loreti, M.: Sessions and pipelines for structured service programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
6. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
7. 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. 222–239. Springer, Heidelberg (2008)
8. Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. Acta Informatica 42(2/3), 191–225 (2005)
9. Gay, S., Vasconcelos, V.T.: Asynchronous functional session types. TR 2007–251. University of Glasgow (May 2007)
10. 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. 22–138. Springer, Heidelberg (1998)
11. Honda, K., Yoshida, N., Carbone, M.: Web Services, Mobile Processes and Types. The Bulletin of the European Association for Theoretical Computer Science 91, 165–185 (2007)
12. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM, New York (2008)
13. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142. Springer, Heidelberg (2008)
14. International Organization for Standardization ISO, 2 UNIversal Financial Industry message scheme (2002), http://www.iso20022.org/index.cfm?item_id=56664#interest
15. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
16. Marlow, S., Jones, S.L.P., Moran, A., Reppy, J.H.: Asynchronous exceptions in Haskell. In: PLDI, pp. 274–285. ACM, New York (2001)
17. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)

18. Vieira, H., Caires, L., Seco, J.: The conversation calculus: A model of service oriented computation. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008)
19. Web Services Choreography Working Group, http://www.w3.org/2002/ws/chor/
20. Xu, J., Romanovsky, A.B., Randell, B.: Concurrent exception handling and resolution in distributed object systems. IEEE Trans. Parallel Distrib. Syst. 11(10), 1019–1032 (2000)
21. Yoshida, N., Vasconcelos, V.T.: Language primitives and type disciplines for structured communication-based programming revisit. ENTCS 171(4), 73–93 (2007)

Global Progress in Dynamically Interleaved Multiparty Sessions*

Lorenzo Bettini[1], Mario Coppo[1], Loris D'Antoni[1], Marco De Luca[1], Mariangiola Dezani-Ciancaglini[1], and Nobuko Yoshida[2]

[1] Dipartimento di Informatica, Università di Torino
[2] Department of Computing, Imperial College London

Abstract. A multiparty session forms a unit of structured interactions among many participants which follow a prescribed scenario specified as a global type signature. This paper develops, besides a more traditional *communication* type system, a novel static *interaction* type system for global progress in dynamically interleaved multiparty sessions.

1 Introduction

Widespread use of message-based communication for developing network applications to combine numerous distributed services has provoked urgent interest in structuring series of interactions to specify and implement program communication-safe software. The actual development of such applications still leaves to the programmer much of the responsibility in guaranteeing that communication will evolve as agreed by all the involved distributed peers. *Multiparty session type discipline* proposed in [12] offers a type-theoretic framework to validate a message-exchange among concurrently running multiple peers in the distributed environment, generalising the existing binary session types [10,11]; interaction sequences are abstracted as a global type signature, which precisely declares how multiple peers communicate and synchronise with each other.

The multiparty sessions aim to retain the powerful dynamic features from the original binary sessions, incorporating features such as recursion and choice of interactions. Among features, *session delegation* is a key operation which permits to rely on other parties for completing specific tasks transparently in a type safe manner. When this mechanism is extended to multiparty interactions engaged in two or more specifications simultaneously, further complex interactions can be modelled. Each multiparty session following a distinct global type can be dynamically *interleaved* by other sessions at runtime either implicitly via communications belonging to different sessions or explicitly via session delegation.

Previous work on multiparty session types [12] has provided a limited progress property ensured only within a single session, ignoring this dynamic nature. More precisely, although the previous system assures that the multiple participants respect the protocol, by checking the types of exchanged messages and the order of communications in a

* The work is partially supported by IST-3-016004-IP-09 SENSORIA, EPSRC GR/T03208, EPSRC EP/F003757 and IST2005-015905 MOBIUS.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 418–433, 2008.

single session, it cannot guarantee a *global progress*, i.e, that a protocol which merges several global scenarios will not get stuck in the middle of a session. This limitation prohibits to ensure a successful termination of a transaction, making the framework practically inapplicable to a large size of dynamically reconfigured conversations.

This paper develops, besides a more traditional *communication* type system (§ 3), a novel static *interaction* type system (§ 4) for global progress in dynamically interleaved multiparty, asynchronous sessions. High-level session processes equipped with global signatures are translated into low-level processes which have explicit senders and receivers. Type-soundness of low-level processes is guaranteed against the local, compositional communication type system.

The new calculus for multiparty sessions offers three technical merits without sacrificing the original simplicity and expressivity in [12]. First it avoids the overhead of global linearity-check in [12]; secondly it provides a more liberal policy in the use of variables, both in delegation and in recursive definitions; finally it implicitly provides each participant of a service with a runtime channel indexed by its role with which he can communicate with all the other participants, permitting also broadcast in a natural way. The use of indexed channels, moreover, permits to define a light-weight interaction type system for global progress.

The interaction type system automatically infers causalities of channels for the low level processes, ensuring the entire protocol, starting from the high-level processes which consist of multiple sessions, does not get stuck at intermediate sessions also in the presence of implicit and explicit session interleaving.

Full definitions and the proofs are at http://www.di.unito.it/ dezani/papers/bcdddy.pdf

2 Syntax and Operational Semantics

Merging Two Conversations: Three-Buyer Protocol. We introduce our calculus through an example, the three-buyer protocol, extending the two-buyer protocol from [12], which includes the new features, session-multicasting and dynamically merging of two conversations. The overall scenario, involving a Seller (S), Alice (A), Bob (B) and Carol (C), proceeds as follows.

1. Alice sends a book title to Seller, then Seller sends back a quote to Alice and Bob. Then Alice tells Bob how much she can contribute.
2. If the price is within Bob's budget, Bob notifies both Seller and Alice he accepts, then sends his address, and Seller sends back the delivery date.
3. If the price exceeds the budget, Bob asks Carol to collaborate together by establishing a new session. Then Bob sends how much Carol must pay, then *delegates* the remaining interactions with Alice and Seller to Carol.
4. If the rest of the price is within Carol's budget, Carol accepts the quote and notifies Alice, Bob and Seller, and continues the rest of the protocol with Seller and Alice transparently, *as if she were Bob*. Otherwise she notifies Alice, Bob and Seller to quit the protocol.

Then multiparty session programming consists of two steps: specifying the intended communication protocols using global types, and implementing these protocols using

processes. The specifications of the three-buyer protocol are given as two separated global types: one is G_a among Alice, Bob and Seller and the other is G_b between Bob and Carol. We write principals with legible symbols though they will actually be coded by numbers: in G_a we have $\mathtt{S} = 3$, $\mathtt{A} = 1$ and $\mathtt{B} = 2$, while in G_b we have $\mathtt{B} = 2$, $\mathtt{C} = 1$.

$G_a =$

1. $\mathtt{A} \longrightarrow \mathtt{S} : \quad \langle \mathsf{string} \rangle.$
2. $\mathtt{S} \longrightarrow \{\mathtt{A},\mathtt{B}\} : \langle \mathsf{int} \rangle.$
3. $\mathtt{A} \longrightarrow \mathtt{B} : \quad \langle \mathsf{int} \rangle.$
4. $\mathtt{B} \longrightarrow \{\mathtt{S},\mathtt{A}\} : \{\mathsf{ok} : \mathtt{B} \longrightarrow \mathtt{S} : \langle \mathsf{string} \rangle.$
5. $\qquad\qquad\qquad\qquad\quad \mathtt{S} \longrightarrow \mathtt{B} : \langle \mathsf{date} \rangle ; \mathsf{end}$
6. $\qquad\qquad\quad \mathsf{quit} : \mathsf{end}\}$

$G_b =$

1. $\mathtt{B} \longrightarrow \mathtt{C} : \langle \mathsf{int} \rangle.$
2. $\mathtt{B} \longrightarrow \mathtt{C} : \langle T \rangle.$
3. $\mathtt{C} \longrightarrow \mathtt{B} : \{\mathsf{ok} : \mathsf{end}, \quad \mathsf{quit} : \mathsf{end}\}.$

$T =$

$\oplus(\{\mathtt{S},\mathtt{A}\},$
$\{\mathsf{ok} : !\langle \mathtt{S}, \mathsf{string} \rangle ; ?\langle \mathtt{S}, \mathsf{date} \rangle ; \mathsf{end},$
$\mathsf{quit} : \mathsf{end}\})$

The types give a global view of the two conversations, directly abstracting the scenario given by the diagram. In G_a, line 1 denotes A sends a string value to S. Line 2 says S multicasts the same integer value to A and B and line 3 says that A sends an integer to B. In lines 4-6 B sends either ok or quit to S and A. In the first case B sends a string to S and receives a date from S, in the second case there are no further communications.

Line 2 in G_b represents the delegation of the capability specified by the action type T of channels (formally defined later) from B to C (note that S and A in T concern the session on a).

We now give the code, associated to G_a and G_b, for S, A, B and C in a "user" syntax formally defined in the following section:

$$\mathrm{S} = \bar{a}[3](y_3).y_3?(\mathit{title});y_3!\langle \mathit{quote} \rangle;y_3\&\{\mathsf{ok} : y_3?(\mathit{address});y_3!\langle \mathit{date} \rangle;\mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$$

$$\mathrm{A} = a[1](y_1).y_1!\langle \texttt{"Title"} \rangle;y_1?(\mathit{quote});y_1!\langle \mathit{quote}\ \mathsf{div}\ 2 \rangle;y_1\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$$

$$\begin{aligned}\mathrm{B} = {} & a[2](y_2).y_2?(\mathit{quote});y_2?(\mathit{contrib});\\ & \text{if } (\mathit{quote} \text{ - } \mathit{contrib} < 100) \text{ then } \ y_2 \oplus \mathsf{ok};y_2!\langle \texttt{"Address"} \rangle;y_2?(\mathit{date});\mathbf{0}\\ & \text{else } \bar{b}[2](z_2).z_2!\langle \mathit{quote} \text{ - } \mathit{contrib} \text{ - } 99 \rangle;z_2!\langle\langle y_2 \rangle\rangle;z_2\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\}\end{aligned}$$

$$\begin{aligned}\mathrm{C} = {} & b[1](z_1).z_1?(x);z_1?((t));\\ & \text{if } (x < 100) \text{ then } z_1 \oplus \mathsf{ok};t \oplus \mathsf{ok};t!\langle \texttt{"Address"} \rangle;t?(\mathit{date});\mathbf{0}\\ & \text{else } z_1 \oplus \mathsf{quit};t \oplus \mathsf{quit};\mathbf{0}\end{aligned}$$

Session name a establishes the session corresponding to G_a. S initiates a session involving three bodies as third participant by $\bar{a}[3](y_3)$: A and B participate as first and second participants by $a[1](y_1)$ and $a[2](y_2)$, respectively. Then S, A and B communicate using the channels y_3, y_1 and y_2, respectively. Each channel $y_\mathtt{p}$ can be seen as a port connecting participant p with all other ones; the receivers of the data sent on $y_\mathtt{p}$ are specified by the global type (this information will be included in the runtime code). The first line of G_a is implemented by the input and output actions $y_3?(\mathit{title})$ and $y_1!\langle \texttt{"Title"} \rangle$. The last line of G_b is implemented by the branching and selection actions $z_2\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$ and $z_1 \oplus \mathsf{ok}$, $z_1 \oplus \mathsf{quit}$.

In B, if the quote minus A's contribution exceeds 100€ (i.e. $\mathit{quote} \text{ - } \mathit{contrib} \geq 100$), another session between B and C is established dynamically through shared name b. The delegation is performed by passing the channel y_2 from B to C (actions $z_2!\langle\langle y_2 \rangle\rangle$ and $z_1?((t))$), and so the rest of the session is carried out by C with S and A. We can

Table 1. Syntax for user-defined processes

$P ::= \bar{u}[n](y).P$	Multicast Request	$\mid$ if e then P else Q Conditional
$\mid u[\mathtt{p}](y).P$	Accept	$\mid P \mid Q$ Parallel
$\mid y!\langle e\rangle;P$	Value sending	$\mid \mathbf{0}$ Inaction
$\mid y?(x);P$	Value reception	$\mid (\nu a)P$ Hiding
$\mid y!\langle\langle z\rangle\rangle;P$	Session delegation	$\mid$ def D in P Recursion
$\mid y?((z));P$	Session reception	$\mid X\langle e,y\rangle$ Process call
$\mid y \oplus l;P$	Selection	
$\mid y\&\{l_i : P_i\}_{i\in I}$	Branching	$e ::= v \mid x$
$u ::= x \mid a$	Identifier	$\mid e$ and $e' \mid$ not $e \ldots$ Expression
$v ::= a \mid$ true $\mid$ false	Value	$D ::= X(x,y) = P$ Declaration

further enrich this protocol with recursive-branching behaviours in interleaved sessions (for example, C can repeatedly negotiate the quote with S as if she were B). What we want to guarantee by static type-checking is that the whole conversation between the four parties preserves progress as if it were a single conversation.

Syntax for Multiparty Sessions. The syntax for processes initially written by the user, called *user-defined processes*, is based on [12]. We start from the following sets: *service names*, ranged over by $a,b,\ldots$ (representing public names of endpoints), *value variables*, ranged over by $x,x',\ldots$, *identifiers* , i.e., service names and variables, ranged over by $u,w,\ldots$, *channel variables*, ranged over by $y,z,t\ldots$, *labels*, ranged over by $l,l',\ldots$ (functioning like method names or labels in labelled records); *process variables*, ranged over by $X,Y,\ldots$ (used for representing recursive behaviour). Then *processes*, ranged over by $P,Q\ldots$, and *expressions*, ranged over by $e,e',\ldots$, are given by the grammar in Table 1.

For the primitives for session initiation, $\bar{u}[n](y).P$ initiates a new session through an identifier u (which represents a shared interaction point) with the other multiple participants, each of shape $u[\mathtt{p}](y).Q_{\mathtt{p}}$ where $1 \leq \mathtt{p} \leq n-1$. The (bound) variable y is the channel used to do the communications. We call p, q,... (ranging over natural numbers) the *participants* of a session. Session communications (communications that take place inside an established session) are performed using the next three pairs of primitives: the sending and receiving of a value; the session delegation and reception (where the former delegates to the latter the capability to participate in a session by passing a channel associated with the session); and the selection and branching (where the former chooses one of the branches offered by the latter). The rest of the syntax is standard from [11].

Global Types. A *global type*, ranged over by $G,G',..$ describes the whole conversation scenario of a multiparty session as a type signature. Its grammar is given below:

Global	$G ::= \mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \langle U\rangle.G'$	Exchange $U ::= S \mid T$
	$\mid \mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \{l_i : G_i\}_{i\in I}$	Sorts $S ::=$ bool $\mid \ldots \mid G$
	$\mid \mu\mathbf{t}.G \mid \mathbf{t} \mid$ end	

We simplify the syntax in [12] by eliminating channels and parallel compositions, while preserving the original expressivity (see § 5).

Table 2. Runtime syntax: the other syntactic forms are as in Table 1

$P ::= c!\langle\{\mathtt{p}_k\}_{k\in K}, e\rangle; P$	Value sending	$\mid c \oplus \langle\{\mathtt{p}_k\}_{k\in K}, l\rangle; P$	Selection
$\mid c?(\mathtt{p}, x); P$	Value reception	$\mid c\&(\mathtt{p}, \{l_i : P_i\}_{i\in I})$	Branching
$\mid c!\langle\langle \mathtt{p}, c'\rangle\rangle; P$	Session delegation	$\mid (\nu s)P$	Hiding session
$\mid c?((\mathtt{q}, y)); P$	Session reception	$\mid s : h$	Named queue
		$\mid \ldots$	
$c ::= y \mid s[\mathtt{p}]$			Channel
$m ::= (\mathtt{q}, \{\mathtt{p}_k\}_{k\in K}, v) \mid (\mathtt{q}, \mathtt{p}, s[\mathtt{p}']) \mid (\mathtt{q}, \{\mathtt{p}_k\}_{k\in K}, l)$			Message in transit
$h ::= m \cdot h \mid \varnothing$			Queue

The global type $\mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \langle U\rangle.G'$ says that participant $\mathtt{p}$ multicasts a message of type U to participants $\mathtt{p}_k$ ($k \in K$) and then interactions described in G' take place. *Exchange types* $U, U', \ldots$ consist of *sorts* types $S, S', \ldots$ for values (either base types or global types), and *action* types $T, T', \ldots$ for channels (discussed in §3). Type $\mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \{l_i : G_i\}_{i\in I}$ says participant $\mathtt{p}$ multicasts one of the labels l_i to participants $\mathtt{p}_k$ ($k \in K$). If l_j is sent, interactions described in G_j take place. Type $\mu\mathbf{t}.G$ is a recursive type, assuming type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way, i.e. type variables only appear under some prefix. We take an *equi-recursive* view of recursive types, not distinguishing between $\mu\mathbf{t}.G$ and its unfolding $G\{\mu\mathbf{t}.G/\mathbf{t}\}$ [18] (§21.8). We assume that G in the grammar of sorts is closed, i.e., without free type variables. Type end represents the termination of the session. We often write $\mathtt{p} \to \mathtt{p}'$ for $\mathtt{p} \to \{\mathtt{p}'\}$.

Runtime Syntax. User defined processes equipped with global types are executed through a translation into runtime processes. The runtime syntax (Table 2) differs from the syntax of Table 1 since the input/output operations (including the delegation ones) specify the sender and the receiver, respectively. Thus, $c!\langle\{\mathtt{p}_k\}_{k\in K}, e\rangle$ sends a value to all the participants in $\{\mathtt{p}_k\}_{k\in K}$; accordingly, $c?(\mathtt{p}, x)$ denotes the intention of receiving a value from the participant $\mathtt{p}$. The same holds for delegation/reception (but the receiver is only one) and selection/branching.

We call $s[\mathtt{p}]$ a *channel with role*: it represents the channel of the participant $\mathtt{p}$ in the session s. We use c to range over variables and channels with roles. As in [12], in order to model TCP-like asynchronous communications (message order preservation and sender-non-blocking), we use the queues of messages in a session, denoted by h; a message in a queue can be a value message, $(\mathtt{q}, \{\mathtt{p}_k\}_{k\in K}, v)$, indicating that the value v was sent by the participant $\mathtt{q}$ and the recipients are all the participants in $\{\mathtt{p}_k\}_{k\in K}$; a channel message (delegation), $(\mathtt{q}, \mathtt{p}', s[\mathtt{p}])$, indicating that $\mathtt{q}$ delegates to $\mathtt{p}'$ the role of $\mathtt{p}$ on the session s (represented by the channel with role $s[\mathtt{p}]$); and a label message, $(\mathtt{q}, \{\mathtt{p}_k\}_{k\in K}, l)$ (similar to a value message). The empty queue is denoted by $\varnothing$. With some abuse of notation we will write $h \cdot m$ to denote that m is the last element included in h and $m \cdot h$ to denote that m is the head of h. By $s : h$ we denote the queue h of the session s. In $(\nu s)P$ all occurrences of $s[\mathtt{p}]$ and the queue s are bound. Queues and the channel with role are generated by the operational semantics (described later).

We present the translation of Bob (B) in the three-buyer protocol with the runtime syntax: the only difference is that all input/output operations specify also the sender and the receiver, respectively.

$$\begin{aligned}\mathrm{B} = a[2](y_2).y_2?(3,\mathit{quote});y_2?(1,\mathit{contrib});\\ \text{if } (\mathit{quote}\text{ - }\mathit{contrib} < 100) \text{ then } y_2 \oplus \langle\{1,3\},\mathtt{ok}\rangle;y_2!\langle\{3\},\mathtt{"Address"}\rangle;y_2?(3,\mathit{date});\mathbf{0}\\ \text{else } \bar{b}[2](z_2).z_2!\langle\{1\},\mathit{quote}\text{ - }\mathit{contrib}\text{ - }99\rangle;z_2!\langle\langle 1,y_2\rangle\rangle;z_2\&(1,\{\mathtt{ok}:\mathbf{0},\ \mathtt{quit}:\mathbf{0}\}).\end{aligned}$$

It should be clear from this example that starting from a global type and user-defined processes respecting the global type it is possible to add sender and receivers to each communication obtaining in this way processes written in the runtime syntax.

We call *pure* a process which does not contain message queues.

Operational Semantics. Table 3 shows the basic rules of the process reduction relation $P \longrightarrow P'$. Rule [Link] describes the initiation of a new session among n participants that synchronises over the service name a. The last participant $\bar{a}[n](y_n).P_n$, distinguished by the overbar on the service name, specifies the number n of participants. For this reason we call it the *initiator* of the session. Obviously each session must have a unique initiator. After the connection, the participants will share the private session name s, and the queue associated to s, which is initialized as empty. The variables $y_{\mathtt{p}}$ in each participant $\mathtt{p}$ will then be replaced with the corresponding channel with role, $s[\mathtt{p}]$. The output rules [Send], [Deleg] and [Label] push values, channels and labels, respectively, into the queue of the session s (in rule [Send], $e \downarrow v$ denotes the evaluation of the expression e to the value v). The rules [Recv], [Srec] and [Branch] perform the corresponding complementary operations. Note that these operations check that the sender matches, and also that the message is actually meant for the receiver (in particular, for [Recv], we need to remove the receiving participant from the set of the receivers in order to avoid reading the same message more than once).

Processes are considered modulo structural equivalence, denoted by $\equiv$, and defined by adding the following rules for queues to the standard ones [17]:

$$s : h_1 \cdot (\mathtt{q},\{\mathtt{p}_k\}_{k\in K},z) \cdot (\mathtt{q}',\{\mathtt{p}_k\}_{k\in K'},z') \cdot h_2 \equiv s : h_1 \cdot (\mathtt{q}',\{\mathtt{p}_k\}_{k\in K'},z') \cdot (\mathtt{q},\{\mathtt{p}_k\}_{k\in K},z) \cdot h_2 \quad \text{if } K\cap K' = \emptyset \text{ or } \mathtt{q}\neq\mathtt{q}'$$

$$s : (\mathtt{q},\emptyset,v)\cdot h \equiv s : h \qquad s : (\mathtt{q},\emptyset,l)\cdot h \equiv s : h$$

where z ranges over v, $s[\mathtt{p}]$ and l. The first rule permits rearranging messages when the senders or the receivers are not the same, and also splitting a message for multiple recipients. The last two rules garbage-collect messages that have already been read by all the intended recipients. We use $\longrightarrow^*$ and $\not\longrightarrow$ with the expected meanings.

Table 3. Selected reduction rules

$$a[1](y_1).P_1 \mid \ldots \mid \bar{a}[n](y_n).P_n \longrightarrow (\nu s)(P_1\{s[1]/y_1\} \mid \ldots \mid P_n\{s[n]/y_n\} \mid s : \varnothing) \quad \text{[Link]}$$

$$s[\mathtt{p}]!\langle\{\mathtt{p}_k\}_{k\in K},e\rangle;P \mid s : h \longrightarrow P \mid s : h\cdot(\mathtt{p},\{\mathtt{p}_k\}_{k\in K},v) \quad (e\downarrow v) \quad \text{[Send]}$$

$$s[\mathtt{p}]!\langle\langle\mathtt{q},s'[\mathtt{p}']\rangle\rangle;P \mid s : h \longrightarrow P \mid s : h\cdot(\mathtt{p},\mathtt{q},s'[\mathtt{p}']) \quad \text{[Deleg]}$$

$$s[\mathtt{p}] \oplus \langle\{\mathtt{p}_k\}_{k\in K},l\rangle;P \mid s : h \longrightarrow P \mid s : h\cdot(\mathtt{p},\{\mathtt{p}_k\}_{k\in K},l) \quad \text{[Label]}$$

$$s[\mathtt{p}_j]?(\mathtt{q},x);P \mid s : (\mathtt{q},\{\mathtt{p}_k\}_{k\in K},v)\cdot h \longrightarrow P\{v/x\} \mid s : (\mathtt{q},\{\mathtt{p}_k\}_{k\in K\setminus j},v)\cdot h \quad (j\in K) \quad \text{[Recv]}$$

$$s[\mathtt{p}]?((\mathtt{q},y));P \mid s : (\mathtt{q},\mathtt{p},s'[\mathtt{p}'])\cdot h \longrightarrow P\{s'[\mathtt{p}']/y\} \mid s : h \quad \text{[Srec]}$$

$$s[\mathtt{p}_j]\&(\mathtt{q},\{l_i : P_i\}_{i\in I}) \mid s : (\mathtt{q},\{\mathtt{p}_k\}_{k\in K},l_{i_0})\cdot h \longrightarrow P_{i_0} \mid s : (\mathtt{q},\{\mathtt{p}_k\}_{k\in K\setminus j},l_{i_0})\cdot h \quad (j\in K)\ \ (i_0\in I) \quad \text{[Branch]}$$

3 Communication Type System

The previous section defines the syntax and the global types. This section introduces the communication type system, by which we can check type soundness of the communications which take place inside single sessions.

Types and Typing Rules for Pure Runtime Processes. We first define the local types of pure processes, called *action types*. While global types represent the whole protocol, action types correspond to the communication actions, representing sessions from the view-points of single participants.

$$\begin{array}{llll} \text{Action} & T ::= & !\langle\{\mathtt{p}_k\}_{k\in K}, U\rangle;T & \textit{send} \\ & \mid & ?(\mathtt{p}, U);T & \textit{receive} \\ & \mid & \oplus\langle\{\mathtt{p}_k\}_{k\in K}, \{l_i : T_i\}_{i\in I}\rangle & \textit{selection} \\ & \mid & \&(\mathtt{p}, \{l_i : T_i\}_{i\in I}) & \textit{branching} \\ & \mid & \mu\mathbf{t}.T & \textit{recursive} \\ & \mid & \mathbf{t} & \textit{variable} \\ & \mid & \mathsf{end} & \textit{end} \end{array}$$

The *send type* $!\langle\{\mathtt{p}_k\}_{k\in K}, U\rangle;T$ expresses the sending to all $\mathtt{p}_k$ for $k \in K$ of a value or of a channel of type U, followed by the communications of T. The *selection type* $\oplus\langle\{\mathtt{p}_k\}_{k\in K}, \{l_i : T_i\}_{i\in I}\rangle$ represents the transmission to all $\mathtt{p}_k$ for $k \in K$ of a label l_i chosen in the set $\{l_i \mid i \in I\}$ followed by the communications described by T_i. The *receive* and *branching* are dual and only need one sender. Other types are standard.

The relation between action and global types is formalised by the notion of projection as in [12]. The *projection of G onto* $\mathtt{q}$ ($G \upharpoonright \mathtt{q}$) is defined by induction on G:

$$(\mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \langle U\rangle.G') \upharpoonright \mathtt{q} = \begin{cases} !\langle\{\mathtt{p}_k\}_{k\in K}, U\rangle;(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ ?(\mathtt{p}, U);(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}_k \text{ for some } k \in K, \\ G' \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(\mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \{l_i : G_i\}_{i\in I}) \upharpoonright \mathtt{q} = \begin{cases} \oplus(\{\mathtt{p}_k\}_{k\in K}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}) & \text{if } \mathtt{q} = \mathtt{p} \\ \&(\mathtt{p}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}) & \text{if } \mathtt{q} = \mathtt{p}_k \text{ for some } k \in K \\ G_1 \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \neq \mathtt{p}, \mathtt{q} \neq \mathtt{p}_k \forall k \in K \text{ and} \\ & G_i \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q} \text{ for all } i, j \in I. \end{cases}$$

$$(\mu\mathbf{t}.G) \upharpoonright \mathtt{q} = \mu\mathbf{t}.(G \upharpoonright \mathtt{q}) \quad \mathbf{t} \upharpoonright \mathtt{q} = \mathbf{t} \quad \mathsf{end} \upharpoonright \mathtt{q} = \mathsf{end}.$$

As an example, we list two of the projections of the global types G_a and G_b of the three-buyer protocol:

$$G_a \upharpoonright 3 = ?\langle 1, \mathsf{string}\rangle; !\langle\{1,2\}, \mathsf{int}\rangle; \&(2, \{\mathsf{ok} : ?\langle 2, \mathsf{string}\rangle; !\langle\{2\}, \mathsf{date}\rangle; \mathsf{end}, \mathsf{quit} : \mathsf{end}\})$$
$$G_b \upharpoonright 1 = ?\langle 2, \mathsf{int}\rangle; ?\langle 2, T\rangle; \oplus\langle\{2\}, \{\mathsf{ok} : \mathsf{end}, \mathsf{quit} : \mathsf{end}\}\rangle$$

where $T = \oplus\langle\{1,3\}, \{\mathsf{ok} : !\langle\{3\}, \mathsf{string}\rangle; ?\langle 3, \mathsf{date}\rangle; \mathsf{end},\ \mathsf{quit} : \mathsf{end}\}\rangle$.

The typing judgements for expressions and pure processes are of the shape:

$$\Gamma \vdash e : S \text{ and } \Gamma \vdash P \triangleright \Delta$$

where Γ is the *standard environment* which associates variables to sort types, service names to global types and process variables to pairs of sort types and action types; Δ is the *session environment* which associates channels to action types. Formally we define:

$$\Gamma ::= \emptyset \mid \Gamma, u : S \mid \Gamma, X : S\ T \text{ and } \Delta ::= \emptyset \mid \Delta, c : T$$

Table 4. Selected typing rules for pure processes

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright n \quad n = \mathrm{pn}(G)}{\Gamma \vdash \bar{u}[n](y).P \triangleright \Delta} \lfloor \textsc{MCast} \rfloor \qquad \frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p}}{\Gamma \vdash u[\mathtt{p}](y).P \triangleright \Delta} \lfloor \textsc{MAcc} \rfloor$$

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle \{\mathtt{p}_k\}_{k \in K}, e \rangle; P \triangleright \Delta, c : !\langle \{\mathtt{p}_k\}_{k \in K}, S \rangle; T} \lfloor \textsc{Send} \rfloor \qquad \frac{\Gamma, x : S \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c?(\mathtt{q}, x); P \triangleright \Delta, c : ?(\mathtt{q}, S); T} \lfloor \textsc{Rcv} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle\langle \mathtt{p}, c' \rangle\rangle; P \triangleright \Delta, c : !\langle \mathtt{p}, T' \rangle; T, c' : T'} \lfloor \textsc{Deleg} \rfloor \qquad \frac{\Gamma \vdash P \triangleright \Delta, c : T, y : T'}{\Gamma \vdash c?((\mathtt{q}, y)); P \triangleright \Delta, c : ?(\mathtt{q}, T'); T} \lfloor \textsc{Srec} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta' \quad dom(\Delta) \cap dom(\Delta') = \emptyset}{\Gamma \vdash P \mid Q \triangleright \Delta \cup \Delta'} \lfloor \textsc{Conc} \rfloor$$

assuming that we can write $\Gamma, u : S$ only if u does not occur in Γ, briefly $u \notin dom(\Gamma)$ ($dom(\Gamma)$ denotes the domain of Γ, i.e. the set of identifiers which occur in Γ). We use the same convention for $X : S\ T$ and Δ.

Table 4 presents the interesting typing rules for pure processes. Rule $\lfloor$MCast$\rfloor$ permits to type a service initiator identified by u, if the type of y is the n-th projection of the global type G of u and the number of participants in G (denoted by $\mathrm{pn}(G)$) is n. Rule $\lfloor$MAcc$\rfloor$ permits to type the $\mathtt{p}$-th participant identified by u, which uses the channel y, if the type of y is the $\mathtt{p}$-th projection of the global type G of u. The successive six rules associate the input/output processes to the input/output types in the expected way. Note that, according to our notational convention on environments, in rule $\lfloor$Deleg$\rfloor$ the channel which is sent cannot appear in the session environment of the premise, i.e. $c' \notin dom(\Delta) \cup \{c\}$. Rule $\lfloor$Conc$\rfloor$ permits to put in parallel two processes only if their sessions environments have disjoint domains. For example we can derive:

$$\vdash t \oplus \langle \{1,3\}, \mathsf{ok} \rangle; t!\langle \{3\}, \texttt{"Address"} \rangle; t?(3, date); \mathbf{0} \triangleright \{t : T\}$$

where $T = \oplus\langle \{1,3\}, \{\mathsf{ok} : !(\{3\}, \mathsf{string}); ?\langle 3, \mathsf{date} \rangle; \mathsf{end},\ \mathsf{quit} : \mathsf{end}\} \rangle$.

In the typing of the example of the three-buyer protocol the types of the channels y_3 and z_1 are the third projection of G_a and the first projection of G_b, respectively. By applying rule $\lfloor$MCast$\rfloor$ we can then derive $a : G_a \vdash \mathrm{S} \triangleright \emptyset$. Similarly by applying rule $\lfloor$MAcc$\rfloor$ we can derive $b : G_b \vdash \mathrm{C} \triangleright \emptyset$.

Types and Typing Rules for Runtime Processes. We now extend the communication type system to processes containing queues.

Message $\mathtt{T}$ ::=	$!\langle \{\mathtt{p}_k\}_{k \in K}, U \rangle$	*message send*	Generalised $\mathtt{T}$ ::=	T	*action*
|	$\oplus\langle \{\mathtt{p}_k\}_{k \in K}, l \rangle$	*message selection*	|	$\mathtt{T}$	*message*
|	$\mathtt{T}; \mathtt{T}'$	*message sequence*	|	$\mathtt{T}; T$	*continuation*

Message types are the types for queues: they represent the messages contained in the queues. The *message send type* $!\langle \{\mathtt{p}_k\}_{k \in K}, U \rangle$ expresses the communication to all $\mathtt{p}_k$ for $k \in K$ of a value or of a channel of type U. The *message selection type* $\oplus\langle \{\mathtt{p}_k\}_{k \in K}, l \rangle$ represents the communication to all $\mathtt{p}_k$ for $k \in K$ of the label l and $\mathtt{T}; \mathtt{T}'$ represents

sequencing of message types. For example $\oplus\langle\{1,3\},\mathsf{ok}\rangle$ is the message type for the message $(2,\{1,3\},\mathsf{ok})$.

A *generalised type* is either an action type, or a message type, or a message type followed by an action type. Type $\mathtt{T};T$ represents the continuation of the type $\mathtt{T}$ associated to a queue with the type T associated to a pure process. An example of generalised type is $\oplus\langle\{1,3\},\mathsf{ok}\rangle;!\langle\{3\},\mathsf{string}\rangle;?\langle 3,\mathsf{date}\rangle;\mathsf{end}$.

We start by defining the typing rules for single queues, in which the turnstile $\vdash$ is decorated with $\{s\}$ (where s is the session name of the current queue) and the session environments are mappings from channels to message types. The empty queue has empty session environment. Each message adds an output type to the current type of the channel which has the role of the message sender.

In order to type pure processes in parallel with queues, we need to use generalised types in session environments and further typing rules. The more interesting rules are:

$$\frac{\Gamma \vdash P \triangleright \Delta}{\Gamma \vdash_{\emptyset} P \triangleright \Delta}\ \lfloor\textsc{GInit}\rfloor \qquad \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Gamma \vdash_{\Sigma'} Q \triangleright \Delta' \quad \Sigma \cap \Sigma' = \emptyset}{\Gamma \vdash_{\Sigma\cup\Sigma'} P \mid Q \triangleright \Delta * \Delta'}\ \lfloor\textsc{GPar}\rfloor$$

where the judgement $\Gamma \vdash_{\Sigma} P \triangleright \Delta$ means that P contains the queues whose session names are in Σ. Rule $\lfloor\textsc{GInit}\rfloor$ promotes the typing of a pure process to the typing of an arbitrary process, since a pure process does not contain queues. When two arbitrary processes are put in parallel (rule $\lfloor\textsc{GPar}\rfloor$) we need to require that each session name is associated to at most one queue (condition $\Sigma \cap \Sigma' = \emptyset$). In composing the two session environments we want to put in sequence a message type and an action type for the same channel with role. For this reason we define the composition $*$ between local types as:

$$\mathsf{T} * \mathsf{T}' = \begin{cases} \mathsf{T};\mathsf{T}' & \text{if } \mathsf{T} \text{ is a message type,} \\ \mathsf{T}';\mathsf{T} & \text{if } \mathsf{T}' \text{ is a message type,} \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ represents failure of typing. We extend $*$ to session environments as expected:
$\Delta * \Delta' = \Delta \backslash dom(\Delta') \cup \Delta' \backslash dom(\Delta) \cup \{c : \mathsf{T} * \mathsf{T}' \mid c : \mathsf{T} \in \Delta \ \&\ c : \mathsf{T}' \in \Delta'\}$.
Note that $*$ is commutative, i.e. $\Delta * \Delta' = \Delta' * \Delta$. Also if we can derive message types only for channels with roles, we consider the channel variables in the definition of $*$ for session environments since we want to get for example $\{y : \mathsf{end}\} * \{y : \mathsf{end}\} = \bot$. An example of derivable judgement is:

$\vdash_{\{s\}} P \mid s : (3,\{1,2\},\mathsf{ok}) \triangleright \{s[3] : \oplus\langle\{1,2\},\mathsf{ok}\rangle;!\langle\{1\},\mathsf{string}\rangle;?\langle 1,\mathsf{date}\rangle;\mathsf{end}\}$

where $P = s[3]!\langle\{1\},\texttt{"Address"}\rangle;s[3]?(1,date);\mathbf{0}$.

Subject Reduction. Since session environments represent the forthcoming communications, by reducing processes session environments can change. This can be formalised as in [12] by introducing the notion of reduction of session environments, whose rules are:

- $\{s[\mathtt{p}] : !\langle\{\mathtt{p}_k\}_{k\in K},U\rangle;\ T, s[\mathtt{p}_j] : ?(\mathtt{p},U);T'\} \Rightarrow \{s[\mathtt{p}] : !\langle\{\mathtt{p}_k\}_{k\in K\setminus j},U\rangle;\ T, s[\mathtt{p}_j] : T'\}$ if $j \in K$
- $\{s[\mathtt{p}] : T;\oplus\langle\{\mathtt{p}_k\}_{k\in K},\{l_i : T_i\}_{i\in I}\rangle\} \Rightarrow \{s[\mathtt{p}] : T;\oplus\langle\{\mathtt{p}_k\}_{k\in K},l_i\rangle;T_i\}$
- $\{s[\mathtt{p}] : \oplus\langle\{\mathtt{p}_k\}_{k\in K},l\rangle;T, s[\mathtt{p}_j] : \&(\mathtt{p},\{l_i : T_i\}_{i\in I})\} \Rightarrow \{s[\mathtt{p}] : \oplus\langle\{\mathtt{p}_k\}_{k\in K\setminus j},l\rangle;T, s[\mathtt{p}_j] : T_i\}$ if $j \in K$ and $l = l_i$
- $\{s[\mathtt{p}] : !\langle\emptyset,U\rangle;\ T\} \Rightarrow \{s[\mathtt{p}] : T\} \qquad \{s[\mathtt{p}] : \oplus\langle\emptyset,l\rangle;T\} \Rightarrow \{s[\mathtt{p}] : T\}$
- $\Delta \cup \Delta'' \Rightarrow \Delta' \cup \Delta''$ if $\Delta \Rightarrow \Delta'$.

The first rule corresponds to the reception of a value or channel by the participant $\mathtt{p}_j$, the second rule corresponds to the choice of the label l_i and the third rule corresponds to the reception of the label l by the participant $\mathtt{p}_j$. The fourth and the fifth rules garbage collect read messages.

Using the above notion we can state type preservation under reduction as follows:

Theorem 1 (Type Preservation). *If $\Gamma \vdash_\Sigma P \triangleright \Delta$ and $P \longrightarrow^* P'$, then $\Gamma \vdash_\Sigma P' \triangleright \Delta'$ for some Δ' such that $\Delta \Rightarrow^* \Delta'$.*

Note that the communication safety [12, Theorem 5.5] is a corollary of this theorem. Thus the user-defined processes with the global types can safely communicate since their runtime translation is typable by the communication type system.

4 Progress

This section studies progress: informally, we say that a process has the progress property if it can never reach a deadlock state, i.e., if it never reduces to a process which contains open sessions (this amounts to containing channels with roles) and which is irreducible in any inactive context (represented by another inactive process running in parallel).

Definition 1 (Progress). *A process P has the* progress property *if $P \longrightarrow^* P'$ implies that either P' does not contain channels with roles or $P' \mid Q \longrightarrow$ for some Q such that $P' \mid Q$ is well typed and $Q \not\longrightarrow$.*

We will give an interaction type system which ensures that the typable processes always have the progress property.

Let us say that a *channel qualifier* is either a session name or a channel variable. Let c be a channel, its channel qualifier $\ell(c)$ is defined by: (1) if $c = y$, then $\ell(c) = y$; (2) else if $c = s[\mathtt{p}]$, then $\ell(c) = s$. Let Λ, ranged over by λ, denote the set of all service names and all channel qualifiers.

The progress property will be analysed via three finite sets: two sets $\mathscr{N}$ and $\mathscr{B}$ of service names and a set $\mathscr{R} \subseteq \Lambda \cup (\Lambda \times \Lambda)$. The set $\mathscr{N}$ collects the service names which are interleaved following the nesting policy. The set $\mathscr{B}$ collects the service names which can be bound. The Cartesian product $\Lambda \times \Lambda$, whose elements are denoted $\lambda \prec \lambda'$, represents a transitive relation. The meaning of $\lambda \prec \lambda'$ is that an input action involving a channel (qualified by) λ or belonging to service λ could block a communication action involving a channel (qualified by) λ' or belonging to service λ'. Moreover $\mathscr{R}$ includes all channel qualifiers and all service names which do not belong to $\mathscr{N}$ or $\mathscr{B}$ and which occur free in the current process. This will be useful to easily extend $\mathscr{R}$ in the assignment rules, as it will be pointed out below. We call $\mathscr{N}$ *nested service set*, $\mathscr{B}$ *bound service set* and $\mathscr{R}$ *channel relation* (even if only a subset of it is, strictly speaking, a relation). Let us give now some related definitions.

Definition 2. *Let $\mathscr{R} ::= \emptyset \mid \mathscr{R}, \lambda \mid \mathscr{R}, \lambda \prec \lambda'$.*

1. $\mathscr{B} \cup \{e\} = \begin{cases} \mathscr{B} \cup \{a\} & \text{if } e = a \text{ is a session name} \\ \mathscr{B} & \text{otherwise.} \end{cases}$

2. $\mathscr{R} \setminus \lambda = \{\lambda_1 \prec \lambda_2 \mid \lambda_1 \prec \lambda_2 \in \mathscr{R} \;\&\; \lambda_1 \neq \lambda \;\&\; \lambda_2 \neq \lambda\} \cup \{\lambda' \mid \lambda' \in \mathscr{R} \;\&\; \lambda' \neq \lambda\}$
3. $\mathscr{R} \setminus\setminus \lambda = \begin{cases} \mathscr{R} \setminus \lambda & \text{if } \lambda \text{ is minimal in } \mathscr{R} \\ \bot & \text{otherwise.} \end{cases}$
4. $\mathscr{R} \uplus \mathscr{R}' = (\mathscr{R} \cup \mathscr{R}')^+$
5. $\mathsf{pre}(\ell(c), \mathscr{R}) = \mathscr{R} \uplus \{\ell(c)\} \uplus \{\ell(c) \prec \lambda \mid \lambda \in \mathscr{R} \;\&\; \ell(c) \neq \lambda\}$

where $\mathscr{R}^+$ *is the transitive closure of (the relation part of)* $\mathscr{R}$ *and* λ *is* minimal *in* $\mathscr{R}$ *if* $\nexists \lambda' \prec \lambda \in \mathscr{R}$.

Note, as it easy to prove, that $\uplus$ is associative. A channel relation is *well formed* if it is irreflexive, and does not contain cycles. A channel relation $\mathscr{R}$ is *channel free* ($\mathsf{cf}(\mathscr{R})$) if it contains only service names.

In Table 5 we introduce selected rules for the interaction type system. The judgements are of the shape: $\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}$ where Θ is a set of *assumptions* of the shape $X[y] \blacktriangleright \mathscr{R}\ ;\ \mathscr{N}\ ;\ \mathscr{B}$ (for recursive definitions) with the variable y representing the channel parameter of X.

We say that a judgement $\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}$ is *coherent* if: (1) $\mathscr{R}$ is well formed; (2) $\mathscr{R} \cap (\mathscr{N} \cup \mathscr{B}) = \emptyset$. We assume that the typing rules are applicable if and only if *the judgements in the conclusion are coherent*.

We will give now an informal account of the interaction typing rules, through a set of examples. It is understood that all processes introduced in the examples can be typed with the communication typing rules given in the previous section.

Table 5. Selected interaction typing rules

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash \bar{a}[n](y).P \blacktriangleright \mathscr{R}\{a/y\}; \mathscr{N}; \mathscr{B}}\{\textsc{MCast}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash a[\mathrm{p}](y).P \blacktriangleright \mathscr{R}\{a/y\}; \mathscr{N}; \mathscr{B}}\{\textsc{MAcc}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash \bar{a}[n](y).P \blacktriangleright \mathscr{R}\setminus\setminus y; \mathscr{N} \cup \{a\}; \mathscr{B}}\{\textsc{MCastN}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash a[\mathrm{p}](y).P \blacktriangleright \mathscr{R}\setminus\setminus y; \mathscr{N} \cup \{a\}; \mathscr{B}}\{\textsc{MAccN}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad \mathsf{cf}(\mathscr{R}\setminus\setminus y)}{\Theta \vdash \bar{u}[n](y).P \blacktriangleright \mathscr{R}\setminus\setminus y; \mathscr{N}; \mathscr{B} \uplus \{u\}}\{\textsc{MCastB}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad \mathsf{cf}(\mathscr{R}\setminus\setminus y)}{\Theta \vdash u[\mathrm{p}](y).P \blacktriangleright \mathscr{R}\setminus\setminus y; \mathscr{N}; \mathscr{B} \uplus \{u\}}\{\textsc{MAccB}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash c!\langle\{\mathrm{p}_k\}_{k\in K}, e\rangle; P \blacktriangleright \{\ell(c)\} \cup \mathscr{R}; \mathscr{N}; \mathscr{B} \uplus \{e\}}\{\textsc{Send}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash c?(\mathsf{q}, x); P \blacktriangleright \mathsf{pre}(\ell(c), \mathscr{R}); \mathscr{N}; \mathscr{B}}\{\textsc{Rcv}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash c!\langle\langle \mathrm{p}', c'\rangle\rangle; P \blacktriangleright \{\ell(c), \ell(c'), \ell(c) \prec \ell(c')\} \uplus \mathscr{R}; \mathscr{N}; \mathscr{B}}\{\textsc{DeleG}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad \mathscr{R} \subseteq \{\ell(c), y, \ell(c) \prec y\}}{\Theta \vdash c?((\mathsf{q}, y)); P \blacktriangleright \{\ell(c)\}; \mathscr{N}; \mathscr{B}}\{\textsc{SRec}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad \Theta \vdash Q \blacktriangleright \mathscr{R}'; \mathscr{N}'; \mathscr{B}'}{\Theta \vdash P \mid Q \blacktriangleright \mathscr{R} \uplus \mathscr{R}'; \mathscr{N} \cup \mathscr{N}'; \mathscr{B} \cup \mathscr{B}'}\{\textsc{Conc}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad a \notin \mathscr{R} \cup \mathscr{N}}{\Theta \vdash (\nu a)P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \setminus a}\{\textsc{NRes}\}$$

$$\frac{}{\Theta, X[y] \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \vdash X\langle e, c\rangle \blacktriangleright \mathscr{R}\{\ell(c)/y\}; \mathscr{N}; \mathscr{B} \uplus \{e\}}\{\textsc{Var}\}$$

$$\frac{\Theta, X[y] \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \qquad \Theta, X[y] \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \vdash Q \blacktriangleright \mathscr{R}'; \mathscr{N}'; \mathscr{B}'}{\Theta \vdash \mathsf{def}\, X(x, y) = P \,\mathsf{in}\, Q \blacktriangleright \mathscr{R}'; \mathscr{N}'; \mathscr{B}'}\{\textsc{Def}\}$$

The crucial point to prove the progress property is to assure that a process, seen as a parallel composition of single threaded processes and queues, cannot be blocked in a configuration in which:

1. there are no thread ready for a session initialization (i.e. of the form $\bar{a}[n](y).P$ or $a[\mathrm{p}](y).P$). Otherwise the process could be reactivated by providing it with the right partners.
2. all subprocesses are either non-empty queues or processes waiting to perform an input action on a channel whose associated queue does not offer an appropriate message.

Progress inside a single service is assured by the communication typing rules in § 3. This will follow as an immediate corollary of Theorem 2. The channel relation is essentially defined to analyse the interactions between services: this is why in the definition of $\mathsf{pre}(\ell(c),\mathscr{R})$ we put the condition $\ell(c) \neq \lambda$. A basic point is that a loop in $\mathscr{R}$ represents the possibility of a deadlock state. For instance take the processes:

$$P_1 = b[1](y_1).\bar{a}[2](z_2).y_1?(2,x);z_2!\langle 1,\mathsf{false}\rangle;\mathbf{0}$$
$$P_2 = \bar{b}[2](y_2).a[1](z_1).z_1?(2,x');y_2!\langle 1,\mathsf{true}\rangle;\mathbf{0}.$$

In process P_1 we have that an input action on service b can block an output action on service a and this determines $b \prec a$. In process P_2 the situation is inverted, determining $a \prec b$. In $P_1 \mid P_2$ we will then have a loop $a \prec b \prec a$. In fact $P_1 \mid P_2$ reduces to

$$Q \;=\; (\nu s)(\nu r)\;\; (s[1]?(2,x);r[1]!\langle 2,\mathsf{false}\rangle;\mathbf{0} \mid r[2]?(1,x');s[2]!\langle 1,\mathsf{true}\rangle;\mathbf{0})$$

which is stuck. It is easy to see that services a and b have the same types, thus we could change b in a in P_1 and P_2 obtaining P_1' and P_2' with two instances of service a and a relation $a \prec a$. But also $P_1' \mid P_2'$ would reduce to Q. Hence we must forbid also loops on single service names (i.e. the channel relation cannot be reflexive).

Rule $\{\textsc{Rcv}\}$ asserts that the input action can block all other actions in P, while rule $\{\textsc{Send}\}$ simply adds $\ell(c)$ in $\mathscr{R}$ to register the presence of a communication action in P. In fact output is asynchronous, thus it can be always performed. Rule $\{\textsc{Deleg}\}$ is similar to $\{\textsc{Send}\}$ but asserts that a use of $\ell(c)$ must precede a use of $\ell(c')$: the relation $\ell(c) \prec \ell(c')$ needs to be registered since an action blocking $\ell(c)$ also blocks $\ell(c')$.

Three different sets of rules handle service initialisations. In rules $\{\textsc{MCast}\}$-$\{\textsc{MAcc}\}$, which are liberal on the occurrences of the channel y in P, the service name a replaces y in $\mathscr{R}$. Rules $\{\textsc{MCastN}\}$-$\{\textsc{MAccN}\}$ can be applied only if the channel y associated to a is minimal in $\mathscr{R}$ .This implies that once a is initialised in P all communication actions on the channel with role instantiating y must be performed before any input communication action on a different channel in P. The name a is added to the nested service set. Remarkably, via rules $\{\textsc{MCastN}\}$-$\{\textsc{MAccN}\}$ we can prove progress when services are nested, generalising the typing strategy of [6]. The rules $\{\textsc{MCastB}\}$ and $\{\textsc{MAccB}\}$ add u to the bound service set whenever u is a service name. These rules are much more restrictive: they require that y is the only free channel in P and that it is minimal. Thus no interaction with other channels or services is possible. This safely allows u to be a variable (since nothing is known about it before execution except its type) or a restricted name (since no channel with role can be made inaccessible at runtime by a restriction on u). Note that rule $\{\textsc{Res}\}$ requires that a occurs neither in $\mathscr{R}$ nor in $\mathscr{N}$.

The sets $\mathscr{N}$ and $\mathscr{B}$ include all service names of a process P whose initialisations is typed with $\{\text{MCASTN}\}$-$\{\text{MACCN}\}$, $\{\text{MCASTB}\}$-$\{\text{MACCB}\}$, respectively. Note that for a service name which will replace a variable this is assured by the (conditional) addition of e to $\mathscr{B}$ in the conclusion of rule $\{\text{SEND}\}$. The sets $\mathscr{N}$ and $\mathscr{B}$ are used to assure, via the coherence condition $\mathscr{R} \cap (\mathscr{N} \cup \mathscr{B}) = \emptyset$, that *all* participants to the same service are typed either by the first two rules or by the remaining four. This is crucial to assure progress. Take for instance the processes P_1 and P_2 above. If we type the session initialisation on b using rule $\{\text{MACCN}\}$ or $\{\text{MACCB}\}$ in P_1 and rule $\{\text{MCAST}\}$ in P_2 no inconsistency would be detected. But rule $\{\text{CONC}\}$ does not type $P_1 \mid P_2$ owing to the coherence condition. Instead if we use $\{\text{MACC}\}$ in P_1, we detect the loop $a \prec b \prec a$. Note that we could not use $\{\text{MCASTN}\}$ or $\{\text{MCASTB}\}$ for b in P_2 since y_2 is not minimal.

Rules $\{\text{MCASTN}\}$-$\{\text{MACCN}\}$ are useful for typing delegation. An example is process B of the three-buyer protocol, in which the typing of the subprocess

$$z_2!\langle\{1\}, \mathit{quote\ -\ contrib}\ -\ 99\rangle; z_2!\langle\langle 1, y_2\rangle\rangle; z_2\&(1, \{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\})$$

gives $z_2 \prec y_2$. So by using rule $\{\text{MCAST}\}$ we would get first $b \prec y_2$ and then the cycle $y_2 \prec b \prec y_2$. Instead using rule $\{\text{MCASTN}\}$ for b we get in the final typing of B either $\{a\};\{b\};\emptyset$ or $\emptyset;\{a,b\};\emptyset$ according to we use either $\{\text{MCAST}\}$ or $\{\text{MCASTN}\}$ for a.

Rule $\{\text{SREC}\}$ avoids to create a process where two different roles in the same session are put in sequence. Following [23] we call this phenomenon self-delegation. As an example consider the processes

$$P_1 = b[1](z_1).a[1](y_1).y_1!\langle\langle 2, z_1\rangle\rangle; \mathbf{0}$$
$$P_2 = \bar{b}[2](z_2).\bar{a}[2](y_2).y_2?((1,x)); x?(2,w); z_2!\langle 1, \mathsf{false}\rangle; \mathbf{0}$$

and note that $P_1 \mid P_2$ reduces to $(\nu s)(\nu r)(s[1]?(2,w); s[2]!\langle 1, \mathsf{false}\rangle; \mathbf{0})$ which is stuck. Note that $P_1 \mid P_2$ is typable by the communication type system but P_2 is not typable by the interaction type system, since by typing $y_2?((1,x)); x?(2,w); z_2!\langle 1, \mathsf{false}\rangle; \mathbf{0}$ we get $y_2 \prec z_2$ which is forbidden by rule $\{\text{SREC}\}$.

A closed runtime process P is *initial* if it is typable both in the communication and in the interaction type systems. The progress property is assured for all computations that are generated from an initial process.

Theorem 2 (Progress). *All initial processes have the progress property.*

It is easy to verify that the (runtime) version of the three-buyer protocol can be typed in the interaction type system with $\{a\};\{b\};\emptyset$ and $\emptyset;\{a,b\};\emptyset$ according to which typing rules we use for the initialisation actions on the service name a. Therefore we get

Corollary 1. *The three-buyer protocol has the progress property.*

5 Conclusions and Related Work

The programming framework presented in this paper relies on the concept of global types that can be seen as the language to describe the model of the distributed communications, i.e., an abstract high-level view of the protocol that all the participants will have to respect in order to communicate in a multiparty communication. The programmer will then write the program to implement this communication protocol; the system

will use the global types (abstract model) and the program (implementation) to generate a runtime representation of the program which consists of the input/output operations decorated with explicit senders and receivers, according to the information provided in the global types. An alternative way could be that the programmer directly specifies the senders and the receivers in the communication operations as our low-level processes; the system could then infer the global types from the program. Our communication and interaction type systems will work as before in order to check the correctness and the progress of the program. Thus the programmer can choose between a top-down and a bottom-up style of programming, while relying on the same properties checked and guaranteed by the system.

We are currently designing and implementing a modelling and specification language with multiparty session types [19] for the standards of business and financial protocols with our industry collaborators [20,21]. This consists of three layers: the first layer is a global type which corresponds to a signature of class models in UML; the second one is for conversation models where signatures and variables for multiple conversations are integrated; and the third layer includes extensions of the existing languages (such as Java [13]) which implement conversation models. We are currently considering to extend this modelling framework with our type discipline so that we can specify and ensure progress for executable conversations.

Multiparty sessions. The first papers on multiparty session types are [2] and [12]. The work [2] uses a distributed calculus where each channel connects a master end-point and one or more slave endpoints; instead of global types, they solely use (recursion-free) local types. In type checking, local types are projected to binary sessions, so that type safety is ensured using duality, but it loses sequencing information: hence progress in a session interleaved with other sessions is not guaranteed.

The present calculus is an essential improvement from [12]; both processes and types in [12] share a vector of channels and each communication uses one of these channels, while our user processes and global types are simpler and user-friendly without these channels. The global types in [12] have a parallel composition operator, but its projectability from global to local types limits to disjoint senders and receivers; hence it does not increase expressivity.

The present calculus is more liberal than the calculus of [12] in the use of declarations, since the definition and the call of recursive processes are obliged to use the same channel variable in [12]. Similarly the delegation in [12] requires that the same channel is sent and received for ensuring subject reduction, as analysed in [23]. Our calculus solves this issue by having channels with roles, as in [9] (see the example at page 430). As a consequence some recursive processes, which are stuck in [12], are type-sound and reducible in our calculus, satisfying the interaction type system.

Different approaches to the description of service-oriented multiparty communications are taken in [3,4]. In [3], the global and local views of protocols are described in two different calculi and the agreement between these views becomes a bisimulation between processes; [4] proposes a distributed calculus which provides communications either inside sessions or inside locations, modelling merging running sessions. The type-safety and progress in interleaved sessions are left as an open problem in [4].

Progress. The majority of papers on service-oriented calculi only assure that clients are never stuck inside a *single* session, see [1,7,12] for detailed discussions, including comparisons between the session-based and the traditional behavioural type systems of mobile processes, e.g. [22,15]. We only say here that our interaction type system is inspired by deadlock-free typing systems [14,15,22]. In [1,7,12], structured session primitives help to give simpler typing systems for progress.

The first papers considering progress for interleaved sessions required the nesting of sessions in Java [8,6] and SOC [1,16,5]. The present approach significantly improves the binary session system for progress in [7] by treating the following points:

(1) asynchrony of the communication with queues, which enhances progress;
(2) a general mechanism of process recursion instead of the limited permanent accepts;
(3) a more liberal treatment of the channels which can be sent; and
(4) the standard semantics for the reception of channels with roles, which permits to get rid of process sequencing.

None of the previous work had treated progress across interfered, dynamically interleaved multiparty sessions.

Acknowledgements. We thank Kohei Honda and the Concur reviewers for their comments on an early version of this paper and Gary Brown for his collaboration on an implementation of multiparty session types.

References

1. Acciai, L., Boreale, M.: A Type System for Client Progress in a Service-Oriented Calculus. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 642–658. Springer, Heidelberg (2008)
2. Bonelli, E., Compagnoni, A.: Multipoint Session Types for a Distributed Calculus. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 240–256. Springer, Heidelberg (2008)
3. 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)
4. Bruni, R., Lanese, I., Melgratti, H., Tuosto, E.: Multiparty Sessions in SOC. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 67–82. Springer, Heidelberg (2008)
5. Bruni, R., Mezzina, L.G.: A Deadlock Free Type System for a Calculus of Services and Sessions (2008), http://www.di.unipi.it/~bruni/publications/ListOfDrafts.html
6. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous Session Types and Progress for Object-Oriented Languages. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 1–31. Springer, Heidelberg (2007)
7. 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)
8. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)

9. Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. Acta Informatica 42(2/3), 191–225 (2005)
10. Honda, K.: Types for Dyadic Interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
11. 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. 22–138. Springer, Heidelberg (1998)
12. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM, New York (2008)
13. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142. Springer, Heidelberg (2008)
14. Kobayashi, N.: A Partially Deadlock-Free Typed Process Calculus. ACM TOPLAS 20(2), 436–482 (1998)
15. 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)
16. Lanese, I., Vasconcelos, V.T., Martins, F., Ravara, A.: Disciplining Orchestration and Conversation in Service-Oriented Computing. In: SEFM 2007, pp. 305–314. IEEE Computer Society Press, Los Alamitos (2007)
17. Milner, R.: Communicating and Mobile Systems: the π-Calculus. CUP (1999)
18. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
19. Scribble Project, http://www.scribble.org
20. UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme (2002), http://www.iso20022.org
21. Web Services Choreography Working Group. Web Services Choreography Description Language, http://www.w3.org/2002/ws/chor/
22. Yoshida, N.: Graph Types for Monadic Mobile Processes. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 371–386. Springer, Heidelberg (1996)
23. Yoshida, N., Vasconcelos, V.T.: Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In: SecRet 2006. ENTCS, vol. 171, pp. 73–93. Elsevier, Amsterdam (2007)

Normed BPA vs. Normed BPP Revisited*

Petr Jančar, Martin Kot, and Zdeněk Sawa

Center for Applied Cybernetics,
Dept. of Computer Science, Technical University of Ostrava,
17. listopadu 15, 70833 Ostrava-Poruba, Czech Republic
petr.jancar@vsb.cz, martin.kot@vsb.cz, zdenek.sawa@vsb.cz

Abstract. We present a polynomial-time algorithm deciding bisimilarity between a normed BPA process and a normed BPP process. This improves the previously known exponential upper bound by Černá, Křetínský, Kučera (1999). The algorithm relies on a polynomial bound for the "finite-state core" of the transition system generated by the BPP process. The bound is derived from the "prime form" of the underlying BPP system (where bisimilarity coincides with equality); we suggest an original algorithm for the respective transformation.

Keywords: verification, equivalence checking, bisimulation equivalence, Basic Process Algebra, Basic Parallel Processes.

1 Introduction

Decidability and complexity of bisimilarity on various classes of processes is a classical topic in process algebra and concurrency theory; see, e.g., [1, 2] for surveys.

One long-standing open problem is the decidability question for the class PA (process algebra), which comprises "context-free" rewrite systems using both sequential and parallel composition. For the subcase of *normed* PA, a procedure working in doubly-exponential nondeterministic time was shown by Hirshfeld and Jerrum [3].

More is known about the "sequential" subclass called BPA (Basic Process Algebra) and the "parallel" subclass called BPP (Basic Parallel Processes). In the case of BPA, the best known algorithm for deciding bisimilarity seems to have doubly-exponential upper bound [1, 4]; the problem is known to be PSPACE-hard [5]. In the case of BPP, the problem is PSPACE-complete [6, 7]. A polynomial-time algorithm for *normed* BPA was shown in [8] (with an upper bound $O(n^{13})$); more recently, an algorithm with running time in $O(n^8 \mathit{polylog}\ n)$ was shown in [9]. For normed BPP, a polynomial time algorithm was presented in [10] (without a precise complexity analysis), based on so called *prime decompositions*; the upper bound $O(n^3)$ was shown in [11] by another algorithm, based on so called *dd-functions*.

* The authors acknowledge the support by the Czech Ministry of Education, Grant No. 1M0567.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 434–446, 2008.

The most difficult matter in the above mentioned algorithm for normed PA [3] is the case when (a process expressed as) sequential composition is bisimilar with (a process expressed as) parallel composition. A basic (sub)problem of this problem is to analyze when a BPA process is bisimilar with a BPP process. Černá, Křetínský, Kučera [12] have shown that this (sub)problem is decidable in the *normed* case; their suggested algorithm is exponential. Decidability in the general (unnormed) case was shown in [13] (without giving any complexity bound).

In this paper, we revisit the normed case, and we present a *polynomial time* algorithm deciding whether a given normed BPA process α is bisimilar with a given normed BPP process M. The main idea is to derive a polynomial bound for the "finite-state core" of the transition system generated by the BPP process M. To this aim we provide a new algorithm, based on *dd*-functions, with time complexity $O(n^3)$, which transforms a given normed BPP process into a "prime form", where bisimilarity coincides with equality. Such a transformation could be based on the prime decompositions in [10] but with worse complexity (which was, in fact, not analyzed in [10]). If the constructed finite-state core exceeds the derived bound, we answer negatively; otherwise we construct a BPA process α' which is bisimilar with M, and the final step is to decide if BPA processes α and α' are bisimilar. This final step can be handled by referring to [8] or [9]. We also sketch a simple self-contained algorithm which uses the fact that α' is close to a finite-state system. This should lead to a better complexity estimation, though we provide no analysis here.

As a side result, our approach also shows a clear polynomial time algorithm testing if there exists a bisimilar BPA process to a given BPP process. This is an alternative to the respective polynomiality result in [12].

Remark. We hope that the new insight will also help to clarify the general (unnormed) case BPA vs. BPP. E.g., the problem mentioned in the previous paragraph seems to be PSPACE-complete in this case.

This paper has the following structure. After basic definitions in Section 2, we describe a transformation of a normed BPP system into the prime form in Section 3. Section 4 contains the crucial result showing the polynomial bound on the "finite-state core". Section 5 finishes the main polynomiality proof.

2 Definitions

We use $\mathbb{N} = \{0, 1, 2, \ldots\}$ to denote the set of nonnegative integers, and we put $\mathbb{N}_{-1} = \mathbb{N} \cup \{-1\}$.

For a set X, $|X|$ denotes the size of X, X^+ denotes the set of nonempty sequences of elements of X, and $X^* = X^+ \cup \{\varepsilon\}$ where ε is the empty sequence. The length of a sequence $x \in X^*$ is denoted by $|x|$ ($|\varepsilon| = 0$). We use x^k (where $x \in X^*$, $k \in \mathbb{N}$) to denote the sequence $xx \cdots x$ where x is repeated k times (in particular $x^0 = \varepsilon$).

A *labelled transition system* (LTS) is a triple $(S, \mathcal{A}, \longrightarrow)$, where S is a set of *states*, $\mathcal{A}$ is a finite set of *actions*, and $\longrightarrow \subseteq S \times \mathcal{A} \times S$ is a *transition relation*. We

write $s \xrightarrow{a} s'$ instead of $(s,a,s') \in \longrightarrow$ and we extend this notation to elements of $\mathcal{A}^*$ in the natural way. We write $s \longrightarrow s'$ if there is $a \in \mathcal{A}$ such that $s \xrightarrow{a} s'$ and $s \longrightarrow^* s'$ if $s \xrightarrow{w} s'$ for some $w \in \mathcal{A}^*$. We write $s \xrightarrow{w}$ if there is some s' such that $s \xrightarrow{w} s'$.

Let $(S, \mathcal{A}, \longrightarrow)$ be an LTS. A binary relation $\mathcal{R} \subseteq S \times S$ is a *bisimulation* iff for each $(s,t) \in \mathcal{R}$ and $a \in \mathcal{A}$ we have:

- $\forall s' \in S : s \xrightarrow{a} s' \Rightarrow (\exists t' : t \xrightarrow{a} t' \wedge (s',t') \in \mathcal{R})$, and
- $\forall t' \in S : t \xrightarrow{a} t' \Rightarrow (\exists s' : s \xrightarrow{a} s' \wedge (s',t') \in \mathcal{R})$.

Less formally, each transition $s \xrightarrow{a} s'$ can be *matched* by a transition $t \xrightarrow{a} t'$ where $(s',t') \in \mathcal{R}$ and vice versa.

States s and t are *bisimulation equivalent (bisimilar)*, written $s \sim t$, iff they are related by some bisimulation. We can also relate states of two different LTSs by taking their disjoint union.

A *BPA (system)* is given by a context-free grammar in Greibach normal form. Formally it is a triple $\Sigma = (V_\Sigma, \mathcal{A}_\Sigma, \Gamma_\Sigma)$, where V_Σ is a finite set of *variables* (nonterminals), $\mathcal{A}_\Sigma$ is a finite set of *actions* (terminals) and $\Gamma_\Sigma \subseteq V_\Sigma \times \mathcal{A}_\Sigma \times V_\Sigma^*$ is a finite set of *rewrite rules*. We will use $V, \mathcal{A}, \Gamma$ for the sets of variables, actions and rules if the underlying BPA is clear from context. Again, we write $X \xrightarrow{a} \alpha$ instead of $(X,a,\alpha) \in \Gamma$. A *BPA process* is a pair (α, Σ) where Σ is a BPA system and $\alpha \in V^*$; we often write just α when Σ is clear from context. A BPA Σ gives rise to the LTS $\mathcal{S}_\Sigma = (V^*, \mathcal{A}, \longrightarrow)$ where $\longrightarrow$ is induced from the rewrite rules by the following (deduction) rule: if $X \xrightarrow{a} \alpha$ then $X\beta \xrightarrow{a} \alpha\beta$ for every $\beta \in V^*$.

A *BPP (system)* is defined in a similar way, as a triple $\Delta = (V_\Delta, \mathcal{A}_\Delta, \Gamma_\Delta)$. The only difference is the deduction rule for the associated LTS $\mathcal{S}_\Delta$: if $X \xrightarrow{a} \alpha$ then $\gamma X \delta \xrightarrow{a} \gamma\alpha\delta$ for any $\gamma, \delta \in V^*$ (thus *any* occurrence of a variable can be rewritten, not just the first one). It is easy to observe that BPP processes α, β with the same Parikh image (i.e., containing the same number of occurrences of each variable) are bisimilar. Hence BPP processes can be read modulo commutativity of concatenation and interpreted as multisets of variables; in the rest of the paper we interpret BPP processes in this way whenever convenient. This also suggests to identify a BPP system Δ with a *BPP net*, a labelled Petri net in which each place corresponds to a variable and each transition corresponds to a rewrite rule (and thus has a unique input place); we will freely do this in our later considerations.

Formally, a *BPP net* is a tuple $\Delta = (P_\Delta, Tr_\Delta, \mathrm{PRE}_\Delta, F_\Delta, \mathcal{A}_\Delta, l_\Delta)$ where P_Δ is a finite set of *places* (variables), Tr_Δ is a finite set of *transitions*, $\mathrm{PRE}_\Delta : Tr_\Delta \to P_\Delta$ is a function assigning an input place to each transition, $F_\Delta : (P_\Delta \times Tr_\Delta) \to \mathbb{N}$ is a *flow function*, $\mathcal{A}_\Delta$ is a finite set of *actions*, and $l_\Delta : Tr_\Delta \to \mathcal{A}_\Delta$ is a *labelling function*. We will use $P, Tr, \mathrm{PRE}, F, \mathcal{A}, l$ if the underlying BPP net is clear from context. A rewrite rule $p \xrightarrow{a} \alpha$ of Δ is represented by a transition $t \in Tr$ such that $\mathrm{PRE}(t) = p$ and $F(t,p')$ is the number of occurrences of p' in α, for each $p' \in P$.

A BPP process is thus, in fact, a *marking*, i.e. a function $M : P \to \mathbb{N}$ which associates a finite number of *tokens* to each place. Note that p^k represents marking M where all k tokens are in one place p ($M(p) = k$ and $M(p') = 0$ for each $p' \neq p$), p represents marking p^1, and ε represents the *zero marking* ($M(p) = 0$ for all $p \in P$).

A transition t is *enabled* at marking M if $M(\text{PRE}(t)) \geq 1$. An enabled transition t may fire from M, producing a marking M' defined by

$$M'(p) = \begin{cases} M(p) - 1 + F(t,p) & \text{if } p = \text{PRE}(t) \\ M(p) + F(t,p) & \text{otherwise} \end{cases} .$$

This is denoted by $M \xrightarrow{t} M'$; the notation is extended to $M \xrightarrow{\sigma} M'$ for sequences $\sigma \in T^*$. We write $M \xrightarrow{\sigma}$ if $M \xrightarrow{\sigma} M'$ for some M'.

In the above sence, a BPP Δ gives rise to the LTS $\mathcal{S}_\Delta = (\mathcal{M}_\Delta, \mathcal{A}, \longrightarrow)$ where $\mathcal{M}_\Delta$ is the set of all markings (of the respective BPP net), and $M \xrightarrow{a} M'$ iff there is some $t \in Tr$ such that $l(t) = a$ and $M \xrightarrow{t} M'$.

In the rest of the paper we use symbols $\alpha, \beta, \ldots$ for both BPA processes and BPP processes, and $M_1, M_2, \ldots$ only for the latter.

We say that a BPA system Σ (a BPP net Δ) is *normed* iff $\alpha \longrightarrow^* \varepsilon$ for each state α of $\mathcal{S}_\Sigma$ ($\mathcal{S}_\Delta$). We will use nBPA (nBPP) for normed BPA (normed BPP).

Our central problem, denoted nBPA-nBPP-BISIM, is defined as follows:

INSTANCE: A normed BPA-process (α_0, Σ), a normed BPP-process (M_0, Δ).
QUESTION: Is $\alpha_0 \sim M_0$ (in the disjoint union of $\mathcal{S}_\Sigma$ and $\mathcal{S}_\Delta$)?

As the size n of an instance of nBPA-nBPP-BISIM we understand the number of bits needed for its (natural) presentation; in particular we consider the numbers $F(t,p)$ in Δ and the numbers in M_0 to be written in binary.

In the rest of this section we assume a fixed nBPA Σ and a fixed nBPP Δ. By a *state* we generally mean a state in the disjoint union of $\mathcal{S}_\Sigma$ and $\mathcal{S}_\Delta$.

Let α be a state (of $\mathcal{S}_\Sigma$ or $\mathcal{S}_\Delta$). *Norm* of α, denoted $\|\alpha\|$, is the length of the shortest $w \in \mathcal{A}^*$ such that $\alpha \xrightarrow{w} \varepsilon$. Note that this also defines norm $\|X\|$ for each variable (place) X. We now note some obvious properties of norms.

- If $\alpha \neq \varepsilon$ then $\|\alpha\| > 0$ for any state α.
- In each nBPA (or nBPP), there is at least one variable (place) with norm 1.
- If $X \xrightarrow{a} \alpha$ is used for a transition $\beta \xrightarrow{a} \beta'$ then $\|\beta'\| - \|\beta\| = \|\alpha\| - \|X\|$.
- $\|\alpha\beta\| = \|\alpha\| + \|\beta\|$ (for BPP-net representation it induces $\|M_1 + M_2\| = \|M_1\| + \|M_2\|$ where marking $M = M_1 + M_2$ is defined componentwise).
- If $\alpha \sim \beta$ then $\|\alpha\| = \|\beta\|$.
- Let $\alpha_1 \sim \alpha_2$, $w \in \mathcal{A}^*$ and $\alpha_1 \xrightarrow{w} \alpha_1'$. There must be a *matching sequence* $\alpha_2 \xrightarrow{w} \alpha_2'$ such that $\alpha_1' \sim \alpha_2'$ (and thus also $\|\alpha_1'\| = \|\alpha_2'\|$).

Finally we note that all norms $\|X\|$, $\|p\|$ for $X \in V_\Sigma$, $p \in P_\Delta$ can be easily computed in polynomial time ($O(n^3)$) and written in polynomial space ($O(n^2)$).

For two states α_1, α_2 we write $\alpha_1 \longrightarrow_R \alpha_2$ if $\alpha_1 \longrightarrow \alpha_2$ and $\|\alpha_2\| = \|\alpha_1\| - 1$. Such a step is called a *norm-reducing step* and the respective rule (transition)

is also called *norm reducing*. We write $\alpha_1 \longrightarrow_R^* \alpha_2$ if there is a sequence (called *norm reducing sequence*) of norm reducing steps leading from α_1 to α_2. For each variable (place) X there is at least one norm-reducing rule (transition) $X \longrightarrow_R \alpha$.

We finish by a few notions concerning the BPP net Δ.

For a marking M and a set $Q \subseteq P$ we define $\|M\|_Q$ as the length of the shortest $w \in \mathcal{A}^*$ such that $M \xrightarrow{w} M'$ where $M'(p) = 0$ for all $p \in Q$.

A place $p \in P$ is *unbounded* in (M_0, Δ) iff for each $c \in \mathbb{N}$ there is a marking M' such that $M_0 \longrightarrow^* M'$ and $M'(p) > c$.

We define $Tok(M) = \sum_{p \in P} M(p)$ and $Car(M) = \{p \in P \mid M(p) \geq 1\}$.

A place p is called a *single final place*, an SF-place, if all transitions that take a token from p are of the form $p \xrightarrow{a} p^k$ (i.e., they can only put tokens back to p). It is easy to see that $\|p\| = 1$ for every SF-place p (since Δ is normed). We say that p is a non-SF-place if it is not an SF-place.

3 Normed BPP Systems in the Prime Form

We say that a *BPP system* Δ is *in the prime form* iff bisimilarity coincides with identity on the generated LTS, i.e., $M \sim M'$ iff $M = M'$.

One way to transform a normed BPP system Δ into an equivalent Δ' in the prime form can be based on the algorithm in [10] which computes certain prime decompositions of BPP-variables (i.e., BPP-net places); it is a polynomial algorithm whose precise complexity has not been analyzed. We use another transformation, which is based on the *dd*-functions and is achieved by an algorithm with time complexity in $O(n^3)$.

In [11], the algorithm from [6] was applied to normed processes. Given a normed BPP system $\Delta = (P, Tr, \text{PRE}, F, \mathcal{A}, l)$, the algorithm finishes in time $O(n^3)$ and constructs a partition $\{T_1, T_2, \ldots, T_m\}$ of the set of transitions such that

$$M \sim M' \text{ iff } d_i(M) = d_i(M') \text{ for all } i = 1, 2, \ldots, m$$

where $d_i(M)$ is the distance to disabling T_i (i.e., the length of the shortest w such that $M \xrightarrow{w} M'$ and in M' all $t \in T_i$ are disabled). Moreover, each class T_i is characterized by the pair (a_i, δ_i) where a_i is the label of all $t \in T_i$ and $\delta_i = (\delta_{i1}, \delta_{i2}, \ldots, \delta_{im})$ is a vector in $(\mathbb{N}_{-1})^m$ such that the following holds for any M, M':

$$\text{if } M \xrightarrow{t} M' \text{ for } t \in T_i \text{ then } d(M') = d(M) + \delta_i$$

where $d(M)$ denotes the vector $(d_1(M), d_2(M), \ldots, d_m(M))$. The type (a_i, δ_i) determines T_i since $(a_i, \delta_i) \neq (a_j, \delta_j)$ for $i \neq j$. For convenience, we will say *transition (of the type)* t_i when meaning any transition $t \in T_i$.

Remark. Space $O(n)$ is sufficient for writing each element of a vector δ_i. There are $O(n)$ such elements in δ_i and $O(n)$ vectors. It follows that space $O(n^3)$ is sufficient for writing all pairs (a_i, δ_i) in binary.

Due to normedness, for every class T_i there is at least one transition t_j which decreases d_i (whenever enabled in M, which also entails $d_i(M) > 0$); this is concisely captured by the next proposition.

Proposition 1. $\forall i \exists j : \delta_{ji} = -1$.

We say that t_i is a *key transition* if it decreases some component of d, i.e. some d_j. Formally we define

$$\mathrm{KEY} = \{i \mid \delta_{ij} = -1 \text{ for some } j\}\,.$$

Proposition 2. $\forall i \in \mathrm{KEY} : \delta_{ii} = -1$.

Proof. If t_i (an element of T_i) decreases some d_j then for each M there is the greatest ℓ such that $M \xrightarrow{(t_i)^\ell}$. The last firing of t_i necessarily decreases d_i. Hence $\delta_{ii} = -1$. □

Thus for each $i \in \mathrm{KEY}$, $d_i(M)$ is the greatest ℓ such that $M \xrightarrow{(t_i)^\ell}$. (A shortest way to disable t_i is to fire it as long as possible.)

We say that t_i *reduces* t_j iff $\delta_{ij} = -1$. Formally we define the following relation RED on KEY:

$$\text{for } i, j \in \mathrm{KEY} \text{ we put } i \text{ RED } j \text{ iff } \delta_{ij} = -1\,.$$

Proposition 3. RED *is an equivalence relation.*

Proof. Reflexivity follows from Proposition 2.
To show symmetricity, assume $i, j \in \mathrm{KEY}$ (so $\delta_{ii} = \delta_{jj} = -1$) such that $\delta_{ij} = -1$ but $\delta_{ji} \geq 0$. Then firing t_j from M with $d_i(M) > 0$ as long as possible results in M' with $d_j(M') = 0$ and $d_i(M') > 0$. Thus $M' \xrightarrow{t_i}$, which is a contradiction since d_j can not be decreased.

Transitivity follows similarly: Suppose i RED j and j RED k but $\neg(i \text{ RED } k)$. So all $\delta_{ii}, \delta_{jj}, \delta_{kk}, \delta_{ij}, \delta_{ji}, \delta_{jk}, \delta_{kj}$ are -1 but $\delta_{ik} \geq 0$. Starting from M with $d_k(M) > 0$, we fire t_i as long as possible and thus get M' with $d_i(M') = d_j(M') = 0$ and $d_k(M') > 0$. Thus $M' \xrightarrow{t_k}$, which is a contradiction since d_j can not be decreased. □

Theorem 4. *There is an algorithm, with time complexity in $O(n^3)$, which transforms a given normed BPP system Δ into Δ' in the prime form, and any given state (marking) M of Δ into M' of Δ' such that $M \sim M'$.*

Proof. In the first phase we compute the partition $\{T_1, T_2, \ldots, T_m\}$ as discussed above. We put $Q_i = \mathrm{PRE}(T_i)$ (where $\mathrm{PRE}(T_i) = \{\mathrm{PRE}(t) \mid t \in T_i\}$) and note that $d_i(M) = \|M\|_{Q_i}$. We now easily verify that $Q_i = Q_j$ for $i, j \in \mathrm{KEY}$ iff i RED j (and so j RED i).

The crucial idea is that Δ' will have a place p_C for each class C of the equivalence RED. For any M of Δ, the number $M'(p_C)$ will be equal to $\|M\|_{Q_i}$ for each $i \in C$.

For every $i \in \mathrm{KEY}$ we add a transition t'_i in Δ' such that $\mathrm{PRE}(t'_i) = p_C$ where $i \in C$; t'_i is labelled with a_i and it realizes the (nonnegative) change on the other places $p_{C'}$ according to δ_i (restricted to KEY).

A non-key transition t_i (with $\delta_i \geq (0, 0, \ldots, 0)$) is enabled precisely when a (key) transition decreasing d_i is enabled (recall Proposition 1). Thus for each p_C where C contains j with $\delta_{ji} = -1$ we add a transition t with label a_i and $\mathrm{PRE}(t) = p_C$ which (gives a token back to p_C and) realizes the change δ_i (restricted to KEY). □

In the following text we only consider BPP systems in the prime form.

4 A Bound on the Number of the "Not-All-in-One-SF" Markings

In this subsection we prove the following theorem.

Theorem 5. *Assume a normed BPA system Σ, with the set V of variables, and a normed BPP system Δ in the prime form, with the set P of places. The number of markings M of Δ such that $\alpha \sim M$ for some $\alpha \in V^+$ and M does not have all tokens in one SF-place is at most $4n^2$, where $n = \max\{|V|, |P|\}$.*

We start with a simple observation and then we bound the total number of tokens in the markings mentioned in the theorem.

Proposition 6. *If $A\alpha \sim M$ where $\alpha \in V^*$ and $|Car(M)| \geq 2$ then $\|A\| \geq 2$.*

Proof. From M with $|Car(M)| \geq 2$ we can obviously perform two different norm-reducing steps resulting in two different, and thus nonbisimilar, markings. On the other hand, any $A\alpha$ with $\|A\| = 1$ has a single outcome (namely α) of any norm-reducing step. □

Proposition 7. *If $|Car(M)| \geq 2$ and $\alpha \sim M$ for $\alpha \in V^+$ then $Tok(M) \leq |V|$.*

Proof. In fact, we prove a stronger proposition. To this aim, we order the variables from V into a sequence $A_1, A_2, \ldots, A_{|V|}$ so that $\|A_i\| \leq \|A_j\|$ for $i \leq j$. We now show the following claim: if $A_i\alpha \sim M$, where $|Car(M)| \geq 2$ (and $\alpha \in V^*$), then $Tok(M) \leq i$.

For the sake of contradiction, suppose a counterexample $A_i\alpha \sim M$, $Tok(M) \geq i{+}1$, for minimal i. Proposition 6 shows that $\|A_i\| \geq 2$, hence also $i \geq 2$ (since necessarily $\|A_1\| = 1$); therefore $Tok(M) \geq i{+}1 \geq 3$. There is thus a norm-reducing step $M \longrightarrow_R M'$ such that $|Car(M')| \geq 2$, $Tok(M') \geq i$. This step is matched by $A_i\alpha \longrightarrow_R A_j\beta\alpha$, $A_j\beta\alpha \sim M'$, where necessarily $\|A_j\| < \|A_i\|$ and thus $j < i$. This is a contradiction with the minimality of our counterexample. □

Since a token from any non-SF-place can be moved to another place (with the total number of tokens non-decreasing), we get the following corollary.

Corollary 8. *If $\alpha \sim M$ then $M(p) \leq |V|$ for every non-SF-place p.*

We now partition the markings in the theorem into four classes:

Class 1. Markings M with all tokens in one (non-SF) place ($|Car(M)| = 1$).
Class 2. Markings M with $|Car(M)| \geq 2$ where at least two different places with norm 1 are reachable; this necessarily means $M \longrightarrow^* M'$ for some M' satisfying $M'(p_1) \geq 1, M'(p_2) \geq 1$ for some $p_1 \neq p_2$ and $\|p_1\| = \|p_2\| = 1$.
Class 3. Markings M with $|Car(M)| \geq 2$ and with exactly one reachable ("sink") place p with norm 1, where p is a non-SF-place.
Class 4. Markings M with $|Car(M)| \geq 2$ and with exactly one reachable ("sink") place p with norm 1, where p is an SF-place.

We will show that each class contains at most n^2 markings by which we prove the theorem. (In fact, our bound is a bit generous, allowing to avoid some technicalities).

Proposition 9. *The number of markings in Class 1 is bounded by* $|V| \cdot |P| \leq n^2$.

Proof. According to Corollary 8 there can be at most $|V|$ tokens in any non-SF-place and there are at most $|P|$ non-SF-places. It follows that Class 1 contains at most $|V| \cdot |P| \leq n^2$ markings. □

Proposition 10. *If* $\alpha \sim M$ *for* M *from Class 2 then* $\alpha = A$ *for some* $A \in V$. *Thus the number of markings in Class 2 is at most* $|V| \leq n$.

Proof. For the sake of contradiction, suppose $A\alpha \sim M$ where $\alpha \in V^+$ and M is from Class 2. We take a counterexample with the minimal length ℓ of a sequence v such that $M \stackrel{v}{\longrightarrow} M'$ where $M'(p_1) \geq 1$, $M'(p_2) \geq 1$ for two different p_1, p_2 with norm 1. We note that $\|A\| \geq 2$ by Proposition 6, and first suppose $\ell > 0$. It is easy to verify that there is a move $M \longrightarrow M''$, matched by $A\alpha \longrightarrow B\beta\alpha$, $B\beta\alpha \sim M''$, where $|Car(M'')| \geq 2$ and the respective length ℓ decreased; this would be a contradiction with the assumed minimality. Thus $\ell = 0$, which means $M(p_1) \geq 1$, $M(p_2) \geq 1$. But then M certainly allows $M \longrightarrow_R^* M_1$, $M \longrightarrow_R^* M_2$ where $\|M_1\| = \|M_2\| = \|\alpha\| \geq 1$ and $M_1 \neq M_2$, and thus $M_1 \not\sim M_2$. On the other hand, $A\alpha$ can offer only α as the result of matching such sequences; hence $A\alpha \not\sim M$. □

Proposition 11. *If* $A\alpha \sim M$ *for* $\alpha \in V^+$ *and* M *from Class 3 or 4 then* $M \longrightarrow_R^* p^{\|\alpha\|}$ *where* p *is the sink place. Thus* $\alpha \sim p^{\|\alpha\|}$.

Proof. We prove the claim by induction on the norm $\|A\|$. Suppose $A\alpha \sim M$ as in the statement. Proposition 6 implies $\|A\| \geq 2$. M necessarily has a token in a place $p' \neq p$ with the least norm greater than 1. Performing a norm-reducing transition with this token corresponds to some $M \longrightarrow_R M'$, and this must be matched by $A\alpha \longrightarrow_R B\beta\alpha$, $B\beta\alpha \sim M'$, where $\|B\| < \|A\|$. Either $|Car(M')| = 1$, in which case necessarily $M' = p^{\|B\beta\alpha\|}$, or $|Car(M')| \geq 2$, and then $M' \longrightarrow_R^* p^{\|\beta\alpha\|}$ due to the induction hypothesis. Since obviously $p^{\|B\beta\alpha\|} \longrightarrow_R^* p^{\|\beta\alpha\|} \longrightarrow_R^* p^{\|\alpha\|}$, we are done. □

Proposition 12. *If $A\alpha \sim M$ where $\|\alpha\| \geq 2$ and M is from Class 3 or 4 then the sink place p is an SF-place. Hence M is from Class 4.*

Proof. For the sake of contradiction, suppose $A\alpha \sim M$ with $\|\alpha\| \geq 2$, M from Class 3, the sink place p thus being a non-SF-place, and assume $\|A\|$ minimal possible; $\|A\| \geq 2$ by Proposition 6.

If there was a step $M \longrightarrow_R M'$ with $|Car(M')| \geq 2$, the matching $A\alpha \longrightarrow_R B\beta\alpha$ would lead to a contradiction with minimality of $\|A\|$. Since $|Car(M)| \geq 2$, the only remaining possibility is the following: $Tok(M) = 2$, $M(p) = 1$ and $M(p') = 1$ where $p' \longrightarrow_R p^k$ for $k = \|A\| + \|\alpha\| - 2 \geq 2$.

Since the sink place p is a non-SF-place, it must be in a cycle C with at least two places. Moving a token along C cannot generate new tokens, due to Corollary 8, so p' is not in C. On the other hand, C contains some p'' with $\|p''\| = 2$. Starting in M, we can move the token from p to p'', the norm being greater than $\|M\| = \|A\alpha\|$ along the way. For the resulting M' we obviously have $M' \longrightarrow_R^* M''$ for M'' satisfying $M''(p'') = 1$ and $\|M''\| = \|\alpha\|$. $A\alpha$ can match this only by reaching α but $\alpha \sim p^{\|\alpha\|}$ according to Proposition 11 and thus $\alpha \not\sim M''$. □

We can thus have $A\alpha \sim M$ for M from Class 3 only when $\|\alpha\| \leq 1$, and it is thus easy to derive the following corollary.

Corollary 13. *The number of markings in Class 3 is at most $|V|^2 \leq n^2$.*

Proposition 14. *The number of markings in Class 4 is at most $|V| \cdot |P| \leq n^2$.*

Proof. Let $A\alpha \sim M$ for M from Class 4, p being the respective SF-sink place. Using Proposition 11, we derive $\alpha \sim I^k$ where $k = \|\alpha\|$ and $I \in V$, $I \sim p$ (such I must exist since $M \longrightarrow^* p$). Thus $AI^k \sim M$ but $AI^k \not\sim I^m$ for any m since $I^m \sim p^m$ and $p^m \not\sim M$ (note that $p^m \neq M$ and Δ is in the prime form).

Since $M \longrightarrow_R^* p^m$ for some m, there must be a (shortest) norm-reducing sequence $A \stackrel{w}{\longrightarrow} B\beta$ where $\beta \sim I^{\|\beta\|}$, $B \not\sim I^{\|B\|}$ but all norm-reducing transitions $B \stackrel{a}{\longrightarrow} \gamma$ satisfy $\gamma \sim I^{\|\gamma\|}$. The sequence $A\alpha \stackrel{w}{\longrightarrow} B\beta\alpha$ (where $B\beta\alpha \sim BI^{\|\beta\alpha\|}$) must be matched by some $M \stackrel{v}{\longrightarrow} M'$ where M' does not have all tokens in p but every norm-reducing transition from M' results in M'' with all tokens in p; it follows that M' has a single token (so we have at most $|P|$ possibilities for M').

This easily implies that there are at most $|V| \cdot |P| \leq n^2$ markings in Class 4. □

5 Problem nBPA-nBPP-BISIM Is in PTIME

We first note that if moving a token along a cycle C in a BPP system Δ generates new tokens in a place p and C is reachable (markable) from M_0 then p is *primarily unbounded* (in M_0). Any place which is unbounded is either primarily unbounded, or *secondarily unbounded*, which means reachable from a primarily unbounded place. Thus any unbounded place has at least one corresponding *pumping cycle*.

We now characterize when there is no nBPA bisimilar with a given nBPP. We say that SF-place p is *growing* if there is a transition $p \xrightarrow{a} p^k$ for $k \geq 2$.

Lemma 15. *For (M_0, Δ), Δ being a normed BPP in the prime form, there is no normed BPA process (α_0, Σ) such that $\alpha_0 \sim M_0$, iff one of the following conditions holds:*

1. *a non-SF-place is unbounded,*
2. *$M_0 \longrightarrow^* M$ with $|Car(M)| \geq 2$ and $M(p) \geq 1$ for some growing SF-place p,*
3. *a non-growing SF-place p is unbounded.*

Proof. If 1. is satisfied then we cannot have $\alpha \sim M_0$ (for any Σ with a [finite] variable set V) due to Corollary 8. If 2. or 3. is satisfied then, for any $c \in \mathbb{N}$, $M_0 \longrightarrow^* M$ with $|Car(M)| \geq 2$ and $Tok(M) > c$. (Any pumping cycle for p in 3. contains $p' \neq p$.) Hence we cannot have $\alpha \sim M_0$ due to Proposition 7.

If none of 1.,2.,3. is satisfied, an appropriate (α, Σ) can be constructed as described below. □

We note that the conditions in Lemma 15 can be checked by straightforward standard algorithms, linear in the size of Δ.

5.1 Construction

Suppose now that a given (M_0, Δ) satisfies none of the conditions 1., 2., 3. in Lemma 15. Thus only growing SF-places can be unbounded. Moreover, if some growing SF-place is reachable from M_0 then $Tok(M_0) = 1$ and each transition sequence reaching p just moves the token into p without creating new tokens on the way.

We can construct the usual reachability graph for M_0, with the exception that the "all-in-one-SF" markings p^k are taken as "frozen" – we construct no successors for them. The thus arising *basic LTS* is necessarily finite, and we can view its states as BPA-variables; each unfrozen marking M is viewed as a variable A_M, with the obvious rewriting rules.

To finish the construction, we introduce a variable I_p for each SF-place p together with appropriate rewriting rules.

More formally, for (M_0, Δ) we could construct nBPA system $\Sigma' = (\mathcal{F} \cup \mathcal{I}, \mathcal{A}, \Gamma')$ where $\mathcal{F} = \{A_M \mid M \in \mathcal{M}_{uf}\}$ (where $\mathcal{M}_{uf} = \{M_1, M_2, \ldots, M_m\}$ is the set of unfrozen markings reachable from M_0), $\mathcal{I} = \{I_p \mid p \in P_{SF}\}$ (where $P_{SF} = \{p_1, p_2, \ldots, p_\ell\}$ is the set of SF-places of Δ), and Γ' contains corresponding rewriting rules.

Note that each rule in Γ' is of one of the following three forms: $A_M \xrightarrow{a} A_{M'}$, $A_M \xrightarrow{a} (I_p)^k$, or $I_p \xrightarrow{a} (I_p)^k$ where $A_M, A_{M'} \in \mathcal{F}$, $I_p \in \mathcal{I}$, and $k \in \mathbb{N}$ (this includes also rules of the form $A_M \xrightarrow{a} \varepsilon$ and $I_p \xrightarrow{a} \varepsilon$). Configuration α'_0 corresponding to M_0 will be A_{M_0} (or $(I_{p_0})^k$ when all k tokens in M_0 are in one SF-place p_0). Note that each configuration α reachable from α'_0 is either of the form A_M or $(I_p)^k$, and we have $(\alpha'_0, \Sigma') \sim (M_0, \Delta)$.

Note that the size of (α'_0, Σ') can be exponential with respect to the size of (M_0, Δ), so we will not construct it explicitly in the algorithm.

Assume an instance of nBPA-nBPP-BISIM, i.e., nBPA (α_0, Σ) and nBPP (M_0, Δ). The polynomial algorithm for nBPA-nBPP-BISIM works as follows.

It first transforms (M_0, Δ) to bisimilar (M'_0, Δ') where Δ' is in the prime form; recall Theorem 4. Then it starts to build the nBPA Σ' for (M', Δ') as described above by building the set $\mathcal{M}_{uf}$ of unfrozen states. If it founds out that the number of elements of $\mathcal{M}_{uf}$ exceeds $4n^2$, where n is the maximum of $\{|V_\Sigma|, |P_{\Delta'}|\}$, then the algorithm stops with the answer $\alpha_0 \not\sim M_0$; this is correct due to Theorem 5.

If the number of elements of $\mathcal{M}_{uf}$ does not exceed $4n^2$, the algorithm finishes the construction of Σ'. However, it does not construct Σ' explicitly but rather a succinct representation of it where right hand sides of rules of the form $(I_p)^k$ are represented as pairs (I_p, k) where k is written in binary. (It can be easily shown that the number of bits of every possible k is in $O(n^2)$ where n is the size of (M'_0, Δ')).

Our aim is to apply the polynomial time algorithm from [8] or [9] to decide if $\alpha_0 \sim \alpha'_0$. However, there is a small technical difficulty since this algorithm expects "usual" nBPA, not nBPA in the succinct form described above. This can be handled by adding special variables I_p^1, I_p^2, I_p^4, I_p^8, $\ldots$ $I_p^{2^m}$ for each $I_p \in \mathcal{I}$ and sufficiently large m (in $O(n^2)$); the rules are adjusted in a straightforward way (note that there will be at most $O(m)$ variables on the right hand side of each rewriting rule after this transformation).

The size of the constructed nBPA is surely polynomial with respect to the size of the original instance of the problem and the algorithm from [8] or [9] can be applied.

So we obtained our main theorem:

Theorem 16. *There is a polynomial-time algorithm deciding whether* $(\alpha_0, \Sigma) \sim (M_0, \Delta)$ *where* Σ *is a normed BPA and* Δ *a normed BPP.*

Since (α'_0, Σ') is in a very special form (it is a finite state system (FS) extended with "SF-tails"), it is in fact not necessary to use the above mentioned general algorithms. Instead we can use a specialized (and probably more efficient) algorithm sketched in the next subsection.

5.2 Specialized Algorithm

The presented algorithm is an adaptation of the standard technique for deciding bisimilarity for a given BPA (or PDA) and a finite-state system used for example in [14, 15].

Assume we have nBPAs (α_0, Σ) and (α'_0, Σ') where (α_0, Σ) is the nBPA from the instance of nBPA-nBPP-BISIM and (α'_0, Σ') is the nBPA described in the previous subsection (with $V_{\Sigma'} = \mathcal{F} \cup \mathcal{I}$) stored using the succinct representation described above (right hand sides of the form $(I_p)^k$ are stored as pairs (I_p, k) with k represented in binary). Let $V_{all} = V_\Sigma \cup V_{\Sigma'}$.

At first we note that the set of configurations from V^*_{all} bisimilar with $(I_p)^k$ where $I_p \in \mathcal{I}$ can be easily characterized. For each $I_p \in \mathcal{I}$ we construct a set $Class(I_p)$ as the maximal subset of V_{all} such that each $X \in Class(I_p)$ can perform exactly the same actions with the same changes on norm as I_p, and can be rewritten only to variables from $Class(I_p)$ (i.e., $X \stackrel{a}{\longrightarrow} \beta$ implies $\beta \in (Class(I_p))^*$, and $I_p \stackrel{a}{\longrightarrow} (I_p)^k$ iff $X \stackrel{a}{\longrightarrow} \beta$ for some $\beta \in (Class(I_p))^*$ such that $\|\beta\| - \|X\| = k - 1$).

The classes $Class(I_p)$ for $I_p \in \mathcal{I}$ can be easily computed in polynomial time. It is not difficult to show that for any $\alpha \in V^*_{all}$ and $I_p \in \mathcal{I}$ we have $\alpha \sim (I_p)^k$ iff $\alpha \in Class(I)^*$ and $\|\alpha\| = k$. Using this fact and precomputed classes $Class(I_p)$, we have a fast (polynomial) test for $\alpha \sim (I_p)^k$.

The crucial observation used in the algorithm is the following. Suppose we want to check if $\alpha \sim A_M$ for some $\alpha \in V^*_{all}$ and $A_M \in \mathcal{F}$ where $\alpha = X\alpha'$ for some $X \in V_{all}$. If $X\alpha' \sim A_M$ then any norm reducing sequence $X\alpha' \longrightarrow^*_R \alpha'$ must be matched by some norm reducing sequence $A_M \longrightarrow^*_R \beta$ such that $\alpha' \sim \beta$. Obviously, β is either of the form $A_{M'}$ (for some $A_{M'} \in \mathcal{F}$) or $(I_p)^k$ (for some $I_p \in \mathcal{I}$). Suppose $\beta = A_{M'}$ (the case $\beta = (I_p)^k$ is similar). Then $\alpha' \sim A_{M'}$. Since $\sim$ is a congruence, we have $XA_{M'} \sim A_M$. On the other hand, if we know that $XA_{M'} \sim A_M$ and $\alpha' \sim A_{M'}$, we know that $X\alpha' \sim A_M$.

By repeating the same approach we can reduce the problem if $\alpha \sim A_M$ to subproblems of testing if $XA_{M'} \sim A_M$, resp. $X(I_p)^k \sim A_M$. Since $\|\alpha\| \neq \|\beta\|$ implies $\alpha \not\sim \beta$, in testing if $X(I_p)^k \sim A_M$ we can consider only those cases where $k = \|A_M\| - \|X\|$. It is obvious that the total number of such subproblems is polynomial with respect to the size of the instance.

The algorithm works by computing the solution for all these subproblems. It approximates from above the set of all such pairs (α, β), where $\alpha \sim \beta$, by computing a fixpoint. It starts with the set of all possible pairs where $\|\alpha\| = \|\beta\|$ and refines it by checking for each pair if it satisfies expansion, i.e., if each transition possible in α is matched by the corresponding transition in β (with respect to the current approximation) and vice versa. (In this checking it also uses the above mentioned test for $\alpha \sim (I_p)^k$).

Obviously the fixed point is reached after polynomial number of iterations. It is not difficult to check that the resulting fixpoint represents the correct set of pairs which then can be used for computing the answer to the original question if $\alpha_0 \sim A_{M_0}$.

References

1. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 545–623. Elsevier, Amsterdam (2001)
2. Srba, J.: Roadmap of infinite results. In: Current Trends In Theoretical Computer Science, The Challenge of the New Century. Formal Models and Semantics, vol. 2, pp. 337–350. World Scientific Publishing Co., Singapore (2004), http://www.brics.dk/~srba/roadmap/

3. 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)
4. Burkart, O., Caucal, D., Steffen, B.: An elementary decision procedure for arbitrary context-free processes. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 423–433. Springer, Heidelberg (1995)
5. Srba, J.: Strong bisimilarity and regularity of Basic Process Algebra is PSPACE-hard. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 716–727. Springer, Heidelberg (2002)
6. Jančar, P.: Strong bisimilarity on Basic Parallel Processes is PSPACE-complete. In: Proc. 18th LiCS, pp. 218–227. IEEE Computer Society, Los Alamitos (2003)
7. Srba, J.: Strong bisimilarity and regularity of Basic Parallel Processes is PSPACE-hard. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 535–546. Springer, Heidelberg (2002)
8. Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial algorithm for deciding bisimilarity of normed context-free processes. Theoretical Computer Science 158, 143–159 (1996)
9. 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)
10. 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)
11. Jančar, P., Kot, M.: Bisimilarity on normed Basic Parallel Processes can be decided in time $O(n^3)$. In: Bharadwaj, R. (ed.) Proceedings of the Third International Workshop on Automated Verification of Infinite-State Systems – AVIS 2004 (2004)
12. Černá, I., Křetínský, M., Kučera, A.: Comparing expressibility of normed BPA and normed BPP processes. Acta Informatica 36, 233–256 (1999)
13. Jančar, P., Kučera, A., Moller, F.: Deciding bisimilarity between BPA and BPP processes. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 159–173. Springer, Heidelberg (2003)
14. Kučera, A., Mayr, R.: Weak bisimilarity between finite-state systems and BPA or normed BPP is decidable in polynomial time. Theoretical Computer Science 270, 667–700 (2002)
15. Kučera, A., Mayr, R.: A generic framework for checking semantic equivalences between pushdown automata and finite-state automata. In: IFIP TCS, pp. 395–408. Kluwer, Dordrecht (2004)

A Rule Format for Associativity

Sjoerd Cranen, MohammadReza Mousavi, and Michel A. Reniers

Department of Computer Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Abstract. We propose a rule format that guarantees associativity of binary operators with respect to all notions of behavioral equivalence that are defined in terms of (im)possibility of transitions, e.g., the notions below strong bisimilarity in van Glabbeek's spectrum. The initial format is a subset of the De Simone format. We show that all trivial generalizations of our format are bound for failure. We further extend the format in a few directions and illustrate its application to several formalisms in the literature. A subset of the format is studied to obtain associativity with respect to graph isomorphism.

1 Introduction

Structural Operational Semantics (SOS) [15] provides a convenient and intuitive way of specifying behavior of systems in terms of states and (labeled) transitions. There are essential properties that must be repeatedly proven for each instance of SOS specification, and the proofs of such properties often follow standard yet tedious lines of reasoning. To facilitate such proofs, many rule formats have been developed for SOS (for an overview, see [1,14]), each with their own syntactic features such as predicates, data stores, and negative formulae. Around each of these formats different meta-theorems have been formulated and proven. These meta-theorems aim at providing a quick and syntactical way of proving the aforementioned properties for the semantics specified by the deduction rules. Examples of such meta-theorems include those concerning congruence of behavioral equivalences [7] and commutativity of operators [12].

Associativity (with respect to a given notion of equivalence) is an interesting property of binary operators, which does not lend itself easily to such syntactic checks. Proofs of associativity are usually much more laborious than proofs of both congruence and commutativity. For example, proofs of congruence, by and large, make use of induction on the proof structure for transitions and are confined to look at the proof-tree *up to depth one*. Thus, they can be performed by looking at deduction rules individually. Proofs of associativity, however, are usually concerned with proof-trees of depth two and hence, if there are n deduction rules for a certain binary operator each with m premises, the number of case-distinctions in its associativity proof are n^m in the worst case (for each deduction rule and each premises, there are n possible deduction rules responsible for the transition mentioned in the premise). This is why in [12, Section 5], we report that our initial attempt to devise a property for associativity did not

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 447–461, 2008.

lead to a concrete rule format. We are not aware of any other rule format for associativity to date. In [10], an abstract 2-categorical framework is presented, which guarantees algebraic properties such as commutativity and associativity. Deriving concrete rule formats from this abstract framework is mentioned as future research in [10].

In this paper we revisit this problem and propose a syntactic rule format that guarantees associativity of a binary operator with respect to any notion of behavioral equivalence that is specified in terms of transitions, e.g., all notions below strong bisimilarity in van Glabeek's spectrum [9,8]. We take the De Simone format [7] as our starting point and add a number of other ingredients, such as predicates [3], to it. Our choice of De Simone format is motivated by the inherent complexity of associativity proofs and is thus aimed to reduce the size and the number of the proof trees as much as possible. The extensions are motivated by practical examples as illustrated in the remainder of this paper. Despite the simple setting of our format, as we show in this paper, our format is widely applicable to most practical examples we encountered so far. Moreover, we show that dropping any of these restrictions jeopardizes the meta-result, even for associativity with respect to the weaker notions of behavioral equivalence, e.g., trace equivalence.

The rest of this paper is structured as follows. We start in Section 2 by presenting some preliminary notions concerning associativity and SOS, used throughout the rest of the paper. In Section 3, we present our basic rule format for associativity and state and prove our associativity meta-theorem with respect to all notions of equivalence that are weaker than (i.e., contain) strong bisimilarity. Section 4 extends our rule format in various directions. In Section 5, we impose some constraints on our format in order to obtain associativity with respect to isomorphism (and all weaker notions). Finally, Section 6 concludes the paper and points out possible directions for future research.

2 Preliminaries

For sake of completeness, we quote standard definitions from concurrency theory and the meta-theory of SOS, which are used in the rest of this paper. A reader familiar with these standard definitions may skip this section altogether.

Definition 1 (Signature and terms). *We assume an infinite set of variables V (with typical members $x, y, x', y', x_i, y_i, \ldots$). A* signature *$\Sigma$ is a set of function symbols (operators) with fixed arities. Functions with zero arity are called* constants. *A* term *$t \in \mathbb{T}(\Sigma)$ is defined inductively as follows:*

- *A variable $x \in V$ is a term.*
- *If $t_1, \ldots, t_n$ are terms then for all $f \in \Sigma$ with arity n, $f(t_1, \ldots, t_n)$ is a term.*

Terms are typically denoted by $t, t', t_i, t'_i, \ldots$. Syntactic equality on terms is denoted by $\equiv$. A closed term *$p \in \mathbb{C}(\Sigma)$, is a term which does not contain any variables. A* substitution *σ is a function from V to $\mathbb{T}(\Sigma)$. The domain of substitution σ is lifted naturally to terms. The range of a* closing substitution *σ is $\mathbb{C}(\Sigma)$.*

Definition 2 (Transition System Specification (TSS), formula). *A* transition system specification *(TSS) is a tuple* (Σ, Rel, D) *where*

- Σ *is a signature*
- Rel *is a set of relation symbols, where for all* $\longrightarrow \in Rel$ *and* $t, t' \in \mathbb{T}(\Sigma)$. *We define that* $(t, t') \in \longrightarrow$ *is a* formula.
- D *is a set of deduction rules. A deduction rule is defined as a tuple* (H, c), *where* H *is a set of formulae and* c *is a formula. The formula* c *is called the* conclusion *and the formulae from* H *are called the* premises *of the deduction rule.*

A formula (also called transition) $(t, t') \in \longrightarrow$ is usually denoted by the more intuitive notation $t \longrightarrow t'$. We refer to t as the *source* and to t' as the *target* of the transition. Mostly, in case there is more than one relation symbol, we use $Rel = \{\xrightarrow{l} \mid l \in L\}$ for some set of *labels* L. A deduction rule (H, c) is usually denoted by $\dfrac{H}{c}$. Such a deduction rule is an (f, l)-*defining* rule of an n-ary operator $f \in \Sigma$ and label $l \in L$ if and only if c is of the form $f(x_1, \cdots, x_n) \xrightarrow{l} t$. A deduction rule is an f-*defining* rule if it is an (f, l)-defining rule for some $l \in L$.

Definition 3 (De Simone format [7]). *A deduction rule is in the* De Simone format *if it is of the form*

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_n) \xrightarrow{l} t}, \quad P$$

where $f \in \Sigma$ *has arity* n, $I \subseteq \{1, \ldots, n\}$ *is a finite set of indices,* P *is a predicate on the labels in the deduction rule and moreover,*

- *for* $1 \leq i < j \leq n$, x_i *and* x_j *are different variables and for* $1 \leq i \leq n$ *and* $j \in I$, x_i *and* y_j *are distinct variables,*
- t *is a term in which the variables from* $\{x_i \mid i \notin I\} \cup \{y_i \mid i \in I\}$ *occur at most once.*

A TSS is in the De Simone format if and only if all of its deduction rules are.

Definition 4 (Provability). *A derived deduction rule* $\dfrac{P}{\;\;c\;\;}$, *also written as the tuple* (P, c), *is provable from a TSS* $\mathcal{T}$, *denoted* $\mathcal{T} \vdash (P, c)$, *when there exists a* proof structure, *i.e., well-founded upwardly branching tree with formulae as nodes and of which*

- *the root node is labeled by* c,
- *if a node is labeled by* ψ *and the labels of the nodes above it form the set* K *then either*
  - $\psi \in P \wedge K = \emptyset$ *or*
  - $\dfrac{K}{\psi}$ *is an instance of a deduction rule of* $\mathcal{T}$.

A formula φ *is provable from* $\mathcal{T}$, *denoted* $\mathcal{T} \vdash \varphi$ *if and only if* $\mathcal{T} \vdash (\emptyset, \varphi)$.

Definition 5 (Bisimulation). *Let $\mathcal{T}$ be a TSS with signature Σ. A relation $\mathcal{R} \subseteq \mathbb{C}(\Sigma) \times \mathbb{C}(\Sigma)$ is a* bisimulation relation *if and only if $\mathcal{R}$ is symmetric and for all $p_0, p_1, p_0' \in \mathbb{C}(\Sigma)$ and $l \in L$*

$$(p_0 \,\mathcal{R}\, p_1 \wedge \mathcal{T} \vdash p_0 \xrightarrow{l} p_0') \Rightarrow \exists_{p_1' \in \mathbb{C}(\Sigma)} (\mathcal{T} \vdash p_1 \xrightarrow{l} p_1' \wedge p_0' \,\mathcal{R}\, p_1').$$

Two terms $p_0, p_1 \in \mathbb{C}(\Sigma)$ are called bisimilar*, denoted by $p_0 \leftrightarrows p_1$ when there exists a bisimulation relation $\mathcal{R}$ such that $p_0 \mathcal{R} p_1$.*

It is easy to check that bisimilarity is indeed an equivalence. Bisimilarity can be extended to open terms by requiring that $t_0 \leftrightarrows t_1$ when $\sigma(t_0) \leftrightarrows \sigma(t_1)$ for all closing substitutions $\sigma : V \to \mathbb{C}(\Sigma)$. In the remainder of this paper, we restrict our attention to the notions of equivalence on *closed terms* that contain strong bisimilarity. However, all our results carry over (without any change) to the notions on *open terms* that contain strong bisimilarity on open terms in the above sense. Another notion of equivalence that we use in the remainder of this paper is isomorphism, as defined below.

Definition 6. *(Isomorphism) Two closed terms p and q are* isomorphic*, denoted by $p \sim_i q$, when there exists a bijective function $h : reach(p) \to reach(q)$ such that $h(p) = q$ and if $h(p_0) = q_0$ and $h(p_1) = q_1$, then $p_0 \xrightarrow{l} p_1$ if and only if $q_0 \xrightarrow{l} q_1$, where $reach(p)$ is the smallest set satisfying $p \in reach(p)$ and if $p' \in reach(p)$ and $p' \xrightarrow{l} q'$, then $q' \in reach(p)$ (i.e., the set of closed terms reachable from p).*

Definition 7 (Associativity). *A binary operator $f \in \Sigma$ is* associative w.r.t. an equivalence $\sim$ *on closed terms if and only if for each $p_0, p_1, p_2 \in \mathbb{C}(\Sigma)$, it holds that $f(p_0, f(p_1, p_2)) \sim f(f(p_0, p_1), p_2)$.*

3 The ASSOC-De Simone Format

In this section, we first specify a limited number of rule types from the De Simone format. Then, we define a number of constraints on the occurrences of rules of such types that guarantee associativity of operators defined by such rules. The format is illustrated by means of a number of examples from the literature. The constraints are obtained by analyzing all proof structures that can be constructed using this restricted set of rule types. The proof of the meta-result, which is a detailed analysis of the proof structures described above, is given next.

3.1 Format

Definition 8 (The ASSOC-De Simone Rule Format). *Consider the following types of rules which are all in the De Simone format. Let $\gamma : L \times L \to L$ be an associative partial function (w.r.t. syntactic equality).*

1_l. *Left-conforming rules*

$$\frac{x \xrightarrow{l} x'}{f(x,y) \xrightarrow{l} f(x',y)}$$

2_l. *Right-conforming rules*

$$\frac{y \xrightarrow{l} y'}{f(x,y) \xrightarrow{l} f(x,y')}$$

3_l. *Left-choice rules*

$$\frac{x \xrightarrow{l} x'}{f(x,y) \xrightarrow{l} x'}$$

4_l. *Right-choice rules*

$$\frac{y \xrightarrow{l} y'}{f(x,y) \xrightarrow{l} y'}$$

5_l. *Left-choice axioms* 6_l. *Right-choice axioms* $7_{(l_0,l_1)}$. *Communicating rules*

$$\frac{}{f(x,y) \xrightarrow{l} x} \qquad \frac{}{f(x,y) \xrightarrow{l} y} \qquad \frac{x \xrightarrow{l_0} x' \quad y \xrightarrow{l_1} y'}{f(x,y) \xrightarrow{\gamma(l_0,l_1)} f(x',y')}$$

A TSS is in the ASSOC-De Simone *format with respect to $f \in \Sigma$ when for each $l \in L$, each f-defining rule is of a type given above and the set of all f-defining rules satisfies the following constraints. (Each proposition P in the following constraints should be read as "there exists a deduction rule of type P in the set of f-defining rules". To avoid repeated uses of parentheses, we assume that $\vee$ and $\wedge$ take precedence over $\Rightarrow$ and $\Leftrightarrow$.)*

1. $5_l \Rightarrow 2_l \wedge 3_l$,
2. $6_l \Rightarrow 1_l \wedge 4_l$,
3. $7_{(l,l')} \Rightarrow (1_l \Leftrightarrow 2_{l'}) \wedge (3_l \Leftrightarrow 4_{l'})$
   $\wedge (2_l \Leftrightarrow 2_{\gamma(l,l')}) \wedge (4_l \Leftrightarrow 4_{\gamma(l,l')}) \wedge (1_{l'} \Leftrightarrow 1_{\gamma(l,l')}) \wedge (3_{l'} \Leftrightarrow 3_{\gamma(l,l')})$,
4. $1_l \wedge 3_l \Leftrightarrow \exists_{l'}\ \gamma(l,l') = l \wedge 7_{(l,l')} \wedge 5_{l'} \wedge 6_{l'}$,
5. $2_l \wedge 4_l \Leftrightarrow \exists_{l'}\ \gamma(l',l) = l \wedge 7_{(l,l')} \wedge 5_{l'} \wedge 6_{l'}$,
6. $(1_l \vee 4_l) \wedge (2_l \vee 3_l) \Rightarrow (5_l \Leftrightarrow 6_l)$.

The types of rules presented above give us a nice starting point while covering many practical applications. These rule types cover all rules that are in the De Simone format with four additional restrictions: Firstly, the target of the conclusion can contain at most one (binary) operator, secondly, the aforementioned operator is the same as the one appearing in the source, thirdly, the labels of the premises and the conclusion coincide (apart from the communicating rule), and finally testing is disallowed.

The main sources of complication in our format are pairs $1_l \wedge 3_l$ and $2_l \wedge 4_l$. If no such pairs are present in the TSS under consideration and moreover, label l generated by 5_l or 6_l cannot synchronize with other labels, i.e., $\nexists_{l,l'}(5_l \vee 6_l) \wedge (7_{(l,l')} \vee 7_{(l',l)})$, then the last three constraints need not be checked. (The last constraint can be dropped in the light of the above-mentioned facts and constraints 1 and 2.) In all practical cases that we have encountered thus far, these conditions hold. Moreover, for each operator with a rule of type $7_{(l_0,l_1)}$, if the presence of an f-defining rule X_l, for $X \in \{1,2,3,4\}$, implies the presence of $X_{l'}$ for all l', then for such an operator, constraint 3 can be simplified to $7_{(l,l')} \Rightarrow ((1_{l''} \Leftrightarrow 2_{l''}) \wedge (3_{l''} \Leftrightarrow 4_{l''}))$. The former assumption on X_l holds for most associative operators in practice but fails for few, such as CSP's parallel composition [16, Chapter 7]. Obviously for operators without defining rules of type $7_{(l_0,l_1)}$, constraint 3 is trivially satisfied.

Theorem 1. *For a TSS in the ASSOC-De Simone format with respect to $f \in \Sigma$, it holds that f is associative for each notion of equivalence $\sim$ containing strong bisimilarity.*

3.2 Examples

In this section we illustrate the ASSOC-De Simone format by means of operators from the literature.

Example 1 (Alternative composition). In most process languages for the description of sequential and parallel systems a form of alternative composition or choice is present. Here we present nondeterministic alternative composition as present in CCS [11] and ACP [4].

$$\frac{x \xrightarrow{l} x'}{x + y \xrightarrow{l} x'} \qquad \frac{y \xrightarrow{l} y'}{x + y \xrightarrow{l} y'}$$

In this example we find deduction rules of types 3 and 4. Therefore, the requirements are met and it can be concluded that alternative composition is associative.

Example 2 (Parallel composition). Another frequently occurring associative operator is parallel composition. It appears in amongst others ACP, CCS, and CCS. Here we discuss parallel composition with communication in the style of ACP [4], for which the others are special cases. It is assumed that an associative (and commutative) partial function γ on labels is given that defines the result of communication and determines the absence or presence of the right-most rule.

$$\frac{x \xrightarrow{l} x'}{x \parallel y \xrightarrow{l} x' \parallel y} \qquad \frac{y \xrightarrow{l} y'}{x \parallel y \xrightarrow{l} x \parallel y'} \qquad \frac{x \xrightarrow{l} x' \quad y \xrightarrow{l'} y'}{x \parallel y \xrightarrow{\gamma(l,l')} x' \parallel y'}$$

Thus, in terms of the types of deduction rules, we have deduction rules of type 1, 2, and 7. Therefore, the requirements are met and it can be concluded that parallel composition is associative.

Example 3 (Disrupt). The disrupt is originally introduced in the language LOTOS [6], where it is used to model for example exception handling. Also, it is used, for example in [2], for the description of mode switches.

$$\frac{x \xrightarrow{l} x'}{x \blacktriangleright y \xrightarrow{l} x' \blacktriangleright y} \qquad \frac{y \xrightarrow{l} y'}{x \blacktriangleright y \xrightarrow{l} y'}$$

Here we see that only deduction rules of types 1 and 4 are present. As a consequence also disrupt is associative.

Example 4 (External choice). The external choice operator $\Box$ from CSP [16] has the following deduction rules:

$$\frac{x \xrightarrow{\tau} x'}{x \Box y \xrightarrow{\tau} x' \Box y} \qquad \frac{y \xrightarrow{\tau} y'}{x \Box y \xrightarrow{\tau} x \Box y'} \qquad \frac{x \xrightarrow{a} x'}{x \Box y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x \Box y \xrightarrow{a} y'}$$

These are of the types 1, 2, 3 and 4, respectively. The constraints of the ASSOC-De Simone format are satisfied since the rules of type 1 and 3 (and 2 and 4) are only there for different labels. Therefore, external choice is associative.

3.3 Proof of Theorem 1

We start with the following auxiliary definition, which will be used in the remainder of our proof.

Definition 9. *(Syntactic Equality Modulo Associativity) Equality modulo associativity of an operator f, denoted by $\simeq_f$, is the smallest reflexive and symmetric relation satisfying $f(p_0, f(p_1, p_2)) \simeq_f f(f(p_0, p_1), p_2)$, for each $p_0, p_1, p_2 \in \mathbb{C}(\Sigma)$.*

The following lemma gives us a stronger thesis from which the theorem follows.

Lemma 1. *An operator $f \in \Sigma$ is associative w.r.t. $\sim$ if $\simeq_f$ is a bisimulation relation.*

Proof. If we prove that $\simeq_f$ is a bisimulation relation, then the theorem follows, because we then have that $f(p_0, f(p_1, p_2)) \underline{\leftrightarrow} f(f(p_0, p_1), p_2)$ and from $\underline{\leftrightarrow} \subseteq \sim$, it follows that $f(p_0, f(p_1, p_2)) \sim f(f(p_0, p_1), p_2)$.

To obtain that $\simeq_f$ is a bisimulation relation, we construct all possible proof structures Pr with $f(p_0, f(p_1, p_2)) \xrightarrow{l} p'$ as conclusion using only rules of the types given in Definition 8. We then show that for each such proof, there is a proof Pr' with $f(f(p_0, p_1), p_2) \xrightarrow{l} p''$ as a conclusion for some p'' such that $p' \simeq_f p''$ and that Pr' uses the premises of Pr. We do the same thing the other way around, so we may conclude that $\simeq_f$ is a bisimulation. By lemma 1 we then conclude that f is associative.

To be able to show the results in a compact way, we introduce the following acronym for proof structures. Let us denote the instantiation of rule r ($r \in \{1, \ldots, 7\}$) with n premises with $r(r_1, r_2, \ldots, r_n)$, where r_i is the rule that is instantiated on premise i. Furthermore, if no rule is instantiated then we denote this with the symbol '$-$', to indicate that no rule is used.

Now the proof for a transition $t \xrightarrow{l} c$ can be denoted with an expression $r \cdot (r_1, r_2, \ldots, r_n)$ when the conclusion of r matches $t \xrightarrow{l} c$ and premise i of rule r matches the conclusion of rule r_i. When the rule used to derive a proof for premises r_i is not relevant, we may write $r \cdot (r_1, \ldots, r_{i-1}, -, \ldots, r_n)$. For readability and when no confusion may arise, we write $r \cdot r_1$ for $r(-, r_1)$, $r(r_1, -)$ or $r \cdot (r_1)$, and write r for $r.(-, -)$, $r \cdot (-)$, or $r \cdot ()$.

Example 5. A proof for the derived deduction rule

$$\frac{x \xrightarrow{l_0} x' \qquad z \xrightarrow{l_1} z'}{f(x, f(y,z)) \xrightarrow{\gamma(l_0,l_1)} f(x', f(y,z'))}$$

is the following

$$\frac{x \xrightarrow{l_0} x' \qquad \dfrac{z \xrightarrow{l_1} z'}{f(y,z) \xrightarrow{l_1} f(y,z')}}{f(x, f(y,z)) \xrightarrow{\gamma(l_0,l_1)} f(x', f(y,z'))} \ .$$

It consists of an instantiation of rule type $7_{(l_0,l_1)}$ and one of type 2_{l_1}, and may therefore be written as $7_{(l_0,l_1)} \cdot (-, 2_{l_1})$ or simply $7_{(l_0,l_1)} \cdot 2_{l_1}$.

Table 1. All proofs with $f(x, f(y,z))$ or $f(f(x,y),z)$ in the source of the conclusion

T_r	T_l	c_r	c_l	further req.
1_l	$1_l \cdot 1_l$	$f(x', f(y,z))$	$f(f(x',y),z)$	
$2_l \cdot 1_l$	$1_l \cdot 2_l$	$f(x, f(y',z))$	$f(f(x,y'),z)$	
$2_l \cdot 2_l$	2_l	$f(x, f(y,z'))$	$f(f(x,y),z')$	
$2_l \cdot 3_l$	$3_l \cdot 2_l$	$f(x,y')$	$f(x,y')$	
$2_l \cdot 4_l$	$7_{(l',l)} \cdot 5_{l'}$	$f(x,z')$	$f(x,z')$	$\gamma(l',l) = l$
$2_l \cdot 5_l$	5_l	$f(x,y)$	$f(x,y)$	
$2_l \cdot 6_l$	$1_l \cdot 5_l$	$f(x,z)$	$f(x,z)$	
$2_{\gamma(l,l')} \cdot 7_{(l,l')}$	$7_{(l,l')} \cdot 2_l$	$f(x, f(y',z'))$	$f(f(x,y'),z')$	
3_l	$3_l \cdot 3_l$	x'	x'	
$4_l \cdot 1_l$	$1_l \cdot 4_l$	$f(y',z)$	$f(y',z)$	
$4_l \cdot 2_l$	$7_{(l',l)} \cdot 6_{l'}$	$f(y,z')$	$f(y,z')$	$\gamma(l',l) = l$
$4_l \cdot 3_l$	$3_l \cdot 4_l$	y'	y'	
$4_l \cdot 4_l$	4_l	z'	z'	
$4_l \cdot 5_l$	$3_l \cdot 6_l$	y	y	
$4_l \cdot 6_l$	6_l	z	z	
$4_{\gamma(l,l')} \cdot 7_{(l,l')}$	$7_{(l,l')} \cdot 4_l$	$f(y',z')$	$f(y',z')$	
5_l	$3_l \cdot 5_l$	x	x	
6_l	$1_l \cdot 6_l$	$f(y,z)$	$f(y,z)$	
$7_{(l,l')} \cdot 1_{l'}$	$1_{\gamma(l,l')} \cdot 7_{(l,l')}$	$f(x', f(y',z))$	$f(f(x',y'),z)$	
$7_{(l,l')} \cdot 2_{l'}$	$7_{(l,l')} \cdot 1_l$	$f(x', f(y,z'))$	$f(f(x',y),z')$	
$7_{(l,l')} \cdot 3_{l'}$	$3_{\gamma(l,l')} \cdot 7_{(l,l')}$	$f(x',y')$	$f(x',y')$	
$7_{(l,l')} \cdot 4_{l'}$	$7_{(l,l')} \cdot 3_l$	$f(x',z')$	$f(x',z')$	
$7_{(l,l')} \cdot 5_{l'}$	$3_l \cdot 1_l$	$f(x',y)$	$f(x',y)$	$\gamma(l,l') = l$
$7_{(l,l')} \cdot 6_{l'}$	$1_l \cdot 3_l$	$f(x',z)$	$f(x',z)$	$\gamma(l,l') = l$
$7_{(l,\gamma(l_0,l_1))} \cdot 7_{(l_0,l_1)}$	$7_{(\gamma(l,l_0),l_1)} \cdot 7_{(l,l_0)}$	$f(x', f(y',z'))$	$f(f(x',y'),z')$	$\gamma(l,\gamma(l_0,l_1)) = \gamma(\gamma(l,l_0),l_1)$

Table 1 shows all the mentioned proofs in an abbreviated way. Column T_r lists the proof structure of proofs with the source of the conclusion $f(x, f(y,z))$, T_l lists the proof structures of those with source of the conclusion $f(f(x,y),z)$. The corresponding targets of the conclusions are listed in columns c_r and c_l.

What is not listed in the table, but what is relevant to the proof of correctness, are the labels of the conclusion transition. These labels are always equal except for the last row, where the labels are $\gamma(l, \gamma(l_0, l_1))$ and $\gamma(\gamma(l, l_0), l_1)$, respectively. By associativity of the function γ these labels are also equal.

We have proved f to be associative if our format guarantees that whenever the rules needed for a proof in the T_r column are present, then the rules needed for the corresponding proof in T_l are present too and the other way around. This is trivially true for those rows in the table where the sets of rules used to construct the proof are the same. In Table 2 we have eliminated these rows. This yields the requirements, listed in the column 'To Prove', on the presence of certain rules. In the column 'Follows From', we indicate which constraints in Definition 8 discharge the proof obligations in the 'To Prove' column.

Table 2. Table 1 without the trivial cases

T_r	T_l	To Prove	Follows From
$2_l \cdot 4_l$	$7_{(l',l)} \cdot 5_l$	$(2_l \wedge 4_l) \Leftrightarrow (\exists_{l'} 7_{(l',l)} \wedge 5_{l'})$	Constraint 5
$2_l \cdot 5_l$	5_l	$5_l \Rightarrow 2_l$	Constraint 1
$2_l \cdot 6_l$	$1_l \cdot 5_l$	$(2_l \wedge 6_l) \Leftrightarrow (1_l \wedge 5_l)$	$\Rightarrow$ Constraints 6, 2 $\Leftarrow$ Constraints 6, 1
$2_{\gamma(l,l')} \cdot 7_{(l,l')}$	$7_{(l,l')} \cdot 2_l$	$7_{(l,l')} \Rightarrow (2_l \Leftrightarrow 2_{\gamma(l,l')})$	Constraint 3
$4_l \cdot 2_l$	$7_{(l',l)} \cdot 6_{l'}$	$(2_l \wedge 4_l) \Leftrightarrow (7_{(l',l)} \wedge 6_{l'})$	Constraint 5
$4_l \cdot 5_l$	$3_l \cdot 6_l$	$(4_l \wedge 5_l) \Leftrightarrow (3_l \wedge 6_l)$	$\Leftarrow$ Constraints 6, 1 $\Rightarrow$ Constraints 6, 2
$4_l \cdot 6_l$	6_l	$6_l \Rightarrow 4_l$	Constraint 2
$4_{\gamma(l,l')} \cdot 7_{(l,l')}$	$7_{(l,l')} \cdot 4_l$	$7_{(l,l')} \Rightarrow (4_l \Leftrightarrow 4_{\gamma(l,l')})$	Constraint 3
5_l	$3_l \cdot 5_l$	$5_l \Rightarrow 3_l$	Constraint 1
6_l	$1_l \cdot 6_l$	$6_l \Rightarrow 1_l$	Constraint 2
$7_{(l,l')} \cdot 1_{l'}$	$1_{\gamma(l,l')} \cdot 7_{(l,l')}$	$7_{(l,l')} \Rightarrow (1_{l'} \Leftrightarrow 1_{\gamma(l,l')})$	Constraint 3
$7_{(l,l')} \cdot 2_{l'}$	$7_{(l,l')} \cdot 1_l$	$7_{(l,l')} \Rightarrow (1_l \Leftrightarrow 2_{l'})$	Constraint 3
$7_{(l,l')} \cdot 3_{l'}$	$3_{\gamma(l,l')} \cdot 7_{(l,l')}$	$7_{(l,l')} \Rightarrow (3_{l'} \Leftrightarrow 3_{\gamma(l,l')})$	Constraint 3
$7_{(l,l')} \cdot 4_{l'}$	$7_{(l,l')} \cdot 3_l$	$7_{(l,l')} \Rightarrow (3_l \Leftrightarrow 4_l)$	Constraint 3
$7_{(l,l')} \cdot 5_{l'}$	$3_l \cdot 1_l$	$7_{(l,l')} \cdot 5_{l'} \Leftrightarrow 3_l \cdot 1_l$	Constraint 4
$7_{(l,l')} \cdot 6_{l'}$	$1_l \cdot 3_l$	$7_{(l,l')} \cdot 6_{l'} \Leftrightarrow 1_l \cdot 3_l$	Constraint 4
$7_{(l,\gamma(l_0,l_1))} \cdot 7_{(l_0,l_1)}$	$7_{(\gamma(l,l_0),l_1)} \cdot 7_{(l,l_0)}$	$7_{(l,\gamma(l_0,l_1))} \wedge 7_{(l_0,l_1)} \Leftrightarrow$ $7_{(\gamma(l,l_0),l_1)} \wedge 7_{(l,l_0)}$	Associativity of γ

3.4 Counter-Examples

The seven basic types of rules that are allowed by the ASSOC-De Simone format are more restrictive than arbitrary rules in the De Simone format in two respects. First, the De Simone format allows for complex terms as the target of the conclusion. However, we only allow for either a variable or applications of the operator being defined (i.e., appearing in the source of the conclusion) on variables. Second, for rules of the first six types, the premise has the same label as the conclusion. In this section, we show that dropping the first restriction jeopardizes our associativity meta-result even with respect to trace equivalence, which is one of the weakest notions of behavioral equivalence. The following two

counter-examples witness that we cannot trivially relax this condition. The first counter-example uses an unary function and the second one uses an associative binary operator (different from the one being defined).

Example 6 (Complex target, I). Consider the terms $f(a, f(a, a))$ and $f(f(a, a), a)$ w.r.t. the following TSS

$$\frac{}{f(x,y) \xrightarrow{a} g(x)} \qquad \frac{}{g(x) \xrightarrow{b} x}$$

The term $f(a, f(a, a))$ can make an a-transition followed by a b-transition and then it deadlocks. However, the $f(f(a, a), a)$ term can make two consecutive ab-traces: $f(f(a, a), a) \xrightarrow{a} g(f(a, a)) \xrightarrow{b} f(a, a) \xrightarrow{a} g(a) \xrightarrow{b} b$.

Example 7 (Complex target, II). Consider the following TSS with the signature containing two constants 0 and a and binary operators f and g (which respectively represent left-merge and parallel-composition operators).

$$(\mathbf{a})\frac{}{a \xrightarrow{a} 0} \qquad (\mathbf{f})\frac{x \xrightarrow{a} x'}{f(x,y) \xrightarrow{a} g(x',y)}$$

$$(\mathbf{g0})\frac{x \xrightarrow{a} x'}{g(x,y) \xrightarrow{a} g(x',y)} \qquad (\mathbf{g1})\frac{y \xrightarrow{a} y'}{g(x,y) \xrightarrow{a} g(x,y')}$$

Consider the terms $f(a, f(0, a))$ and $f(f(a, 0), a)$. The former term can only make an a-transition into $g(0, f(0, a))$ (by the proof structure $\mathbf{f} \cdot \mathbf{a}$); the target of this transition, in turn, deadlocks. The latter term can only make an a-transition into $g(g(0, 0), a)$ (by the proof structure $\mathbf{g0} \cdot \mathbf{g0} \cdot \mathbf{a}$); however the target of this transition can make one further a-transition into $g(g(0, 0), 0)$.

The restriction of the treatment to deduction rules without relabeling (excluding the communicating rule) is not essential, but allows for a simpler presentation of the format. It remains future work to formulate and prove a rule format for associativity in the presence of relabeling.

In Section 4, we extend our format by introducing testing in the premises and predicates.

4 Possible Extensions

In this section, we investigate extensions of our format in various directions.

4.1 Testing Rules

De Simone format does not allow for premises of which the targets do not appear in the target of the conclusion. This phenomenon is called testing in the SOS literature and, albeit disallowed by De Simone format, is of practical relevance, e.g., in modeling predicates. Thus, in this section, we introduce the concept of

testing to our ASSOC-De Simone format to cover these practical cases. The only relevant type of testing which allows us to model predicates is given by the following type of rules. As we demonstrate in Section 4.3, other sorts of testing (predicate) rules can be coded using a combination of rules already present in our ASSOC-De Simone format.

$$8_{(l,l')}.\ \text{Left-choice + test}\quad \frac{x \xrightarrow{l} x' \quad y \xrightarrow{l'} y'}{f(x,y) \xrightarrow{l} x'} \qquad 9_{(l',l)}.\ \text{Right-choice + test}\quad \frac{x \xrightarrow{l'} x' \quad y \xrightarrow{l} y'}{f(x,y) \xrightarrow{l} y'}$$

The constraints we give next for the above two types of testing rules can be generalized to arbitrary testing rules (with relabeling and changing operators).

Definition 10 (The ASSOC-De Simone Rule Format with Testing). *A TSS is in the ASSOC-De Simone format with testing w.r.t. f, when each defining rule is of one of the previously given types, the rules of types $\{1_l, \ldots, 6_l, 7_{(l,l')}\}$ satisfy the constraints of the ASSOC-De Simone format w.r.t. f and moreover, the following constraints hold for the rules of types $8_{(l,l')}$ and $9_{(l,l')}$:*

1. $8_{(l,l')} \wedge X_p \Rightarrow 3_l$ *and* $9_{(l',l)} \wedge X_p \Rightarrow 4_l$, *for each* $X_p \in \{1_{l'}, \ldots, 6_{l'}, 7_{(l_0,l_1)}, | \gamma(l_0,l_1) = l' \wedge l_0 \neq l' \vee l_1 \neq l'\}$,
2. $(8_{(l,l')} \wedge 1_l) \Rightarrow \exists_{l''} \gamma(l'',l) = l \wedge 7_{(l'',l)} \wedge 5_{l''}$ *and* $(9_{(l',l)} \wedge 2_l) \Rightarrow \exists_{l''} \gamma(l'',l) = l \wedge 7_{(l'',l)} \wedge 5_{l''}$,
3. $8_{(l,l')} \wedge 6_l \Rightarrow 5_l$ *and* $9_{(l',l)} \wedge 5_l \Rightarrow 6_l$,
4. $7_{(l_0,l_1)} \Rightarrow (8_{(l_1,l')} \Leftrightarrow 8_{(\gamma(l_0,l_1),l')}) \wedge (8_{(l_0,l')} \Leftrightarrow 9_{(l',l_1)}) \wedge (9_{(l',\gamma(l_0,l_1))} \Leftrightarrow 9_{(l',l_0)})$,
5. $8_{(l,l')} \wedge 8_{(l,l'')} \Rightarrow 8_{(l',l'')} \vee (7_{(l',l'')} \wedge 8_{(l,\gamma(l',l''))})$ *and* $9_{(l',l)} \wedge 9_{(l'',l)} \Rightarrow 9_{(l',l'')} \vee (7_{(l',l'')} \wedge 9_{(\gamma(l',l''),l)})$.

Theorem 2. *For a TSS in the ASSOC-De Simone format with testing w.r.t. $f \in \Sigma$, it holds that f is associative for each notion of equivalence $\sim$ containing strong bisimilarity.*

Due to space limitations, proof of the theorem is omitted. Later in Section 4.3, we show how rules of type 8 and 9 can be used to obtain associativity of operators in the definition of which predicates are involved, e.g., sequential composition operator.

4.2 Changing Operators

In the ASSOC-De Simone format, the only operator that may appear in the target of the conclusion is the same as the operator appearing in the source. This assumption may not hold in practice and is not essential to our meta-result, either. Thus, in this section, we relax this assumption and allow for rules of the following shape.

$$1_l.\ \text{Left-conf. rules}\quad \frac{x \xrightarrow{l} x'}{f(x,y) \xrightarrow{l} g(x',y)} \qquad 2_l.\ \text{Right-conf. rules}\quad \frac{y \xrightarrow{l} y'}{f(x,y) \xrightarrow{l} g(x,y')} \qquad 7_{(l_0,l_1)}.\ \text{Comm. rules}\quad \frac{x \xrightarrow{l_0} x' \quad y \xrightarrow{l_1} y'}{f(x,y) \xrightarrow{\gamma(l_0,l_1)} g(x',y')}$$

Note that if g is taken to be the same as f, then we recover the original types of rules allowed in the ASSOC-De Simone format.

Definition 11 (The ASSOC-De Simone Rule Format with Changing Operators). *Let $\Sigma' \subseteq \Sigma$. A TSS is in the ASSOC-De Simone format with changing operators w.r.t. Σ' when for each $f \in \Sigma'$ the following constraints are satisfied:*

- *the f-defining rules are in the ASSOC-De Simone format (with relaxed targets of conclusions as described above),*
- *for each $l \in L$, the (f,l)-defining rules of type 7 all have the same operator $g \in \Sigma'$ in the target of the conclusion, and there are no (f,l)-defining rules of type 1 or 2 with $f \neq g$, where g is the operator appearing in the target of the conclusion.*

Theorem 3. *For a TSS in the ASSOC-De Simone format with changing operators w.r.t. $\Sigma' \in \Sigma$, it holds that each $f \in \Sigma'$ is associative for each notion of equivalence $\sim$ containing strong bisimilarity.*

Proof. The proof is obtained by adapting Table 1 for those occurrences of rules of types 1, 2 and 7 with a changing operator, say g. The targets of the conclusions, i.e., c_l and c_r then stay the same terms with all occurrences of f replaced by g.

Example 8 (Communication merge). Consider the communication merge operator $\mid$ from ACP [4] with the following deduction rule

$$\frac{x \xrightarrow{l} x' \quad y \xrightarrow{l'} y'}{x \mid y \xrightarrow{\gamma(l,l')} x' \parallel y'}$$

and additionally those of the parallel composition operator from Example 2. Recall that the communication function γ is assumed to be associative. Note that this rule is of type $7_{(l,l')}$. The constraints of the ASSOC-De Simone fromat with changing operators are satisfied. Thus, as a consequence, communication merge is associative.

4.3 Predicates

Assume that a predicate $\mathcal{P}$ is given by deduction rules of the following form.

$$\frac{\mathcal{P}_x}{\mathcal{P}_{f(x,y)}} \qquad \frac{\mathcal{P}_y}{\mathcal{P}_{f(x,y)}} \qquad \frac{\mathcal{P}_x \quad \mathcal{P}_y}{\mathcal{P}_{f(x,y)}}$$

In order to accommodate such predicates in our framework and our format, we use a translation inspired by [17]. The following deduction rules of types $1_{\mathcal{P}}$, $2_{\mathcal{P}}$, and $7_{(\mathcal{P},\mathcal{P})}$, respectively, code predicate $\mathcal{P}$ in the ASSOC-De Simone format.

$$\frac{x \xrightarrow{\mathcal{P}} x'}{f(x,y) \xrightarrow{\mathcal{P}} f(x',y)} \qquad \frac{y \xrightarrow{\mathcal{P}} y'}{f(x,y) \xrightarrow{\mathcal{P}} f(x,y')} \qquad \frac{x \xrightarrow{\mathcal{P}} x' \quad y \xrightarrow{\mathcal{P}} y'}{f(x,y) \xrightarrow{\mathcal{P}} f(x',y')}$$

Other possible types of rules for defining predicates can be coded similarly inside the ASSOC-De Simone rule format. The major difficulty here is in combining predicates with transitions. This can be done in our framework, using either communicating rules (type $7_{(l,l')}$) or testing rules (types $8_{(l,l')}$ or $9_{(l,l')}$). The following example illustrates this.

Example 9 (Sequential Composition). Consider the following deduction rules defining the sequential composition operator.

$$\frac{x \xrightarrow{l} x'}{x \cdot y \xrightarrow{l} x' \cdot y} \qquad \frac{x\downarrow \quad y \xrightarrow{l} y'}{x \cdot y \xrightarrow{l} y'} \qquad \frac{x\downarrow \quad y\downarrow}{x \cdot y \downarrow}$$

The second deduction rule uses the termination predicate as a premise. Translation of the later two deduction rules to a setting without predicates gives

$$\frac{x \xrightarrow{\downarrow} x' \quad y \xrightarrow{l} y'}{x \cdot y \xrightarrow{l} y'} \qquad \frac{x \xrightarrow{\downarrow} x' \quad y \xrightarrow{\downarrow} y'}{x \cdot y \xrightarrow{\gamma(\downarrow,\downarrow)} x' \cdot y'}$$

with $\gamma(\downarrow,\downarrow) = \downarrow$ and undefined otherwise. These rules are of type 1_l, $9_{(\downarrow,l)}$, and $7_{(\downarrow,\downarrow)}$. In Definition 10, the only constraints of which the left-hand-side of the implications hold are 4 and 5. Thus, we only need to check that $(8_{(\downarrow,l')} \Leftrightarrow 8_{(\downarrow,l')}) \wedge (8_{(\downarrow,l')} \Leftrightarrow 9_{(l',\downarrow)}) \wedge (9_{(l',\downarrow)} \Leftrightarrow 9_{(l',\downarrow)})$ and $9_{(\downarrow,\downarrow)} \vee (7_{(\downarrow,\downarrow)} \wedge 9_{(\downarrow,l)})$. The former holds trivially since none of the propositions appearing in the three bi-implications hold. The latter holds, as well, because we have that $7_{(\downarrow,\downarrow)} \wedge 9_{(\downarrow,l)}$. Thus, we can conclude that all constraints of the ASSOC-De Simone format with testing are satisfied and hence sequential composition is associative.

5 Associativity for Isomorphism

Although associativity w.r.t. strong bisimilarity provides us with a strong meta-result that is capable of dealing with all applications in practice, an even stronger result can be obtained, if we prove associativity w.r.t. isomorphism as given in Definition 6.

In this section, we first show that our meta-result does not trivially carry over to the case where isomorphism is considered as the notion of equivalence. Then, we seek extra conditions under which associativity w.r.t. isomorphism indeed holds. The following example shows why the ASSOC-De Simone format cannot be used as is for proving associativity w.r.t. isomorphism.

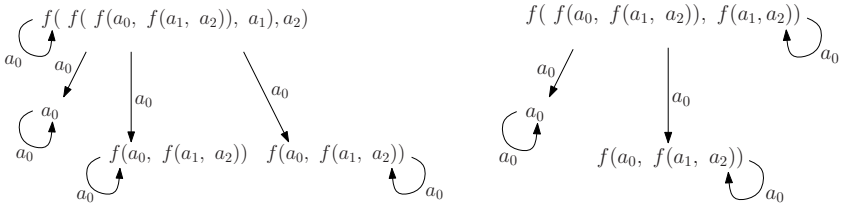
Example 10 (Associativity w.r.t. Isomorphism). Consider the following TSS.

$$(\mathbf{a_i})\frac{}{a_i \xrightarrow{a_i} a_i} \qquad (\alpha,\beta)\frac{x \xrightarrow{\alpha} x' \quad y \xrightarrow{\beta} y'}{f(x,y) \xrightarrow{\gamma(\alpha,\beta)} f(x',y')}$$

$$(\mathbf{la_0})\frac{x \xrightarrow{a_0} x'}{f(x,y) \xrightarrow{a_0} x'} \qquad (\mathbf{ra_0})\frac{y \xrightarrow{a_0} y'}{f(x,y) \xrightarrow{a_0} y'}$$

α	β	$\gamma(\alpha,\beta)$
a_0	a_1	a'
a_1	a_2	a'
a_0	a'	a_0
a'	a_2	a_0

where rule $(\mathbf{a_i})$ is present only for $i = 0, 1, 2$ and rule (α, β) is only defined for all pairs of α and β for which the table on the right provides an entry. Next, the LTS's of the terms $f(\ f(\ f(a_0,\ f(a_1,\ a_2)),\ a_1), a_2) \simeq_f f(\ f(a_0,\ f(a_1,\ a_2)),\ f(a_1, a_2))$ are depicted. Note that these two LTS's are not isomorphic since one of them comprises three states and the other one comprises four states.



Theorem 4 gives sufficient conditions for associativity w.r.t. isomorphism.

Theorem 4. *For a TSS in the ASSOC-De Simone format w.r.t. $f \in \Sigma$ such that, disregarding the labels, the set of all f-defining rules satisfies $(1 \vee 2 \vee 7) \Leftrightarrow \neg(3 \vee 4 \vee 5 \vee 6 \vee 8 \vee 9)$ (all f-defining rules are either of types 1, 2 and 7, or are all of the other types), then f' is associative w.r.t. isomorphism.*

Proof. The proviso of Theorem 4 requires that the f-defining rules are either of the types 1,2,7 or of the types 3,4,5,6,8,9. We call the deduction rules of the first type *f-preserving* and those of the second type *f-eliminating*.

If all f-defining rules are f-preserving, then all states in the LTS of $f(p_0, f(p_1, p_2))$ are of the form $f(q_0, f(q_1, q_2))$. Then, define $h(f(q_0, f(q_1, q_2))) \doteq f(f(q_0, q_1), q_2)$. Then, it is straightforward to check (by consulting the corresponding rows of Table 1) that h is the bijective function satisfying the constraints of Definition 6.

If all f-defining rules are f-eliminating, then define $h(f(p_0, f(p_1, p_2))) \doteq f(f(p_0, p_1), p_2)$ (for the initial state) and $f(p') = p'$ for all nodes reachable from the initial state. Note that the initial state cannot have a self-loop, because the size of the term strictly decreases in each transition. Thus, the above definition gives rise to a (function and a) bijection. It is also straightforward to check that in the rows of Table 1 when the applied rules are all f-eliminating, then the targets of the transitions are syntactically equal and thus our bijection satisfies the constraints of Definition 6.

6 Conclusions

In the context of binary operators specified by means of deduction rules in the well-known De Simone format, we developed a rule format guaranteeing associativity w.r.t. any notion of behavioral equivalence containing strong bisimilarity. The format is adapted for the setting with predicates, testing rules and 'changing operators'. Applicability of the format is shown by means of examples from literature.

We plan to extend the ASSOC-De Simone format to deal with relabeling and negative premises [5]. Another direction for future research is a more general format in the setting of weak equivalences such as branching bisimilarity and weak

bisimilarity. An extension of the ASSOC-De Simone format to deal with a notion of state/date is also anticipated (see [13]).

Acknowledgment. Insightful comments of CONCUR'08 reviewers led to a number of improvements and are gratefully acknowledged.

References

1. Aceto, L., Fokkink, W., Verhoef, C.: Structural Operational Semantics. In: Handbook of Process Algebra, ch. 3, pp. 197–292. Elsevier, Amsterdam (2001)
2. Baeten, J.C.M., Bergstra, J.A.: Mode transfer in process algebra. Technical Report CSR-00-01, Dept. of Computer Science, TU/Eindhoven (2000)
3. Baeten, J.C.M., Verhoef, C.: A congruence theorem for structured operational semantics with predicates. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 477–492. Springer, Heidelberg (1993)
4. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge (1990)
5. Bol, R., Groote, J.F.: The meaning of negative premises in transition system specifications. J. ACM 43(5), 863–914 (1996)
6. Brinksma, E.: A tutorial on LOTOS. In: Proc. of Protocol Specification, Testing and Verification V, pp. 171–194. North-Holland, Amsterdam (1985)
7. de Simone, R.: Higher-level synchronizing devices in MEIJE-SCCS. TCS 37, 245–267 (1985)
8. van Glabbeek, R.J.: The linear time - branching time spectrum I. In: Handbook of Process Algebra, ch.1, pp. 3–99. Elsevier, Amsterdam (2001)
9. van Glabbeek, R.J.: The linear time - branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
10. Hasuo, I., Jacobs, B., Sokolova, A.: The microcosm principle and concurrency in coalgebra. In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962. Springer, Heidelberg (to appear, 2008)
11. Milner, A.J.R.G.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
12. Mousavi, M.R., Reniers, M.A., Groote, J.F.: A syntactic commutativity format for SOS. IPL 93, 217–223 (2005)
13. Mousavi, M.R., Reniers, M.A., Groote, J.F.: Notions of bisimulation and congruence formats for SOS with data. I&C 200(1), 107–147 (2005)
14. Mousavi, M.R., Reniers, M.A., Groote, J.F.: SOS formats and meta-theory: 20 years after. TCS (373), 238–272 (2007)
15. Plotkin, G.D.: A structural approach to operational semantics. JLAP 60, 17–139 (2004)
16. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1997)
17. Verhoef, C.: A congruence theorem for structured operational semantics with predicates and negative premises. Nordic Journal of Computing 2(2), 274–302 (1995)

Deriving Structural Labelled Transitions for Mobile Ambients

Julian Rathke and Paweł Sobociński*

ECS, University of Southampton

Abstract. We present a new labelled transition system (LTS) for the ambient calculus on which ordinary bisimilarity coincides with contextual equivalence. The key feature of this LTS is that it is the fruit of ongoing work on developing a systematic procedure for deriving LTSs in the structural style from the underlying reduction semantics and observability. Notably, even though we have derived our LTS for ambients systematically it compares very favourably with existing transition systems for the same calculus.

Introduction

Since the introduction of the archetypal process calculi, CCS [20], CSP [14] and ACP [2], and the π-calculus [9, 21] some years ago there has been a proliferation of calculus extensions and variants which address assorted computational features. One concern that is often voiced regarding these extended calculi is that their semantics, particularly their labelled transition semantics, are often ad hoc and heavily locally optimised. This state of affairs is unsatisfactory and initial attempts to address the issue were made in [17, 26] where it was proposed that labelled transitions should be *derived* (rather than defined) by means of considering underlying reduction rules for the language and taking labels to be suitably minimal contexts which induce reductions. The rationale for this is that for any new computational feature, its reduction rules are generally easier to define uncontentiously and can be taken to be definitional. Consequently the derived labelled transitions would be given without further design. This approach is appealing but Sewell's early results [26] were limited in their scope. Leifer and Milner generalised the approach with some degree of success [17]. A general definition of contexts-as-labels was provided using the universal property of (relative) pushouts to obtain a suitable notion of minimality. Even so, this work still has its problems, the chief of which is that the derived labelled transition systems are not presented in an inductive manner and are therefore often difficult to describe and reason with.

It is easy to lose sight of the fact that the original intention of structural operational semantics [23] and labelled transition systems [20] was to provide an inductive definition of the reduction relation for a language. Their subsequent use as points of comparison of interaction in bisimulation equivalences has allowed focus to drift away from inductively defined labelled transition systems and on to

* Research partially supported by EPSRC grant EP/D066565/1.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 462–476, 2008.

labels as the contextually observable parts of interaction. Our general research goal is to provide a method by which structurally defined labelled transition systems can be derived from an underlying reduction semantics. For this derived transition system, bisimulation equivalence must also correspond to a contextually defined equivalence. This task is difficult and we have begun by evaluating our ideas for simple process calculi. The results of such an experiment for the π-calculus appear in [24].

In this paper we apply our method to the ambient calculus of Cardelli and Gordon [7]. The ambient calculus has enjoyed success as a foundational model of distributed processes. It essentially comprises hierarchically arranged processes which can migrate, as well as dynamically modify the structure of their location. Our interest is not in the ambient calculus as a model of distributed computing per se but simply as a small calculus with an interesting set of reduction rules for which it has thus far proven difficult to provide a definitive labelled transition system and bisimulation equivalence [5, 18]. Our purpose is not necessarily to improve or undermine the existing labelled transition systems but to systematically derive one.

The approach we take is to consider the underlying ground rewrite rules of the language as structural rewrites supplied with suitable ground parameters. For any term which partially matches the left-hand-side (LHS) of a rewrite, we identify the parameters supplied by the term to the match. A label represents the remaining structure of the LHS of the rewrite rule along with the missing parameters which will be supplied by an interacting context. This separation of the structure of the rewrite and the parameters to the rule allows us to build our labelled transition systems in three steps: we define the process-view transitions, whose main purpose is to provide an inductively defined reduction relation, then the context-view transitions, which allows for a context to supply parameters to an interaction, and finally rules to combine them. Technically, we make use of the simply typed λ-calculus as a meta-language for abstracting parameters.

Structure of the paper. We present the syntax and semantics of the ambient calculus, along with a suitable contextually defined equivalence, in the next section. We then give an account of our method of deriving labelled transitions and show its instantiation for the ambient calculus in Section 2. Section 3 lists the properties of the LTS: bisimulation equivalence is proved to be sound for reduction barbed congruence and, after the addition of Honda-Tokoro [15, 25] style rules to account for unobservability of certain actions, complete. We include a comparison with related work in Section 4 and close with some concluding remarks regarding future work. Due to space constraints, detailed proofs have been omitted.

1 Ambients: Syntax, Metasyntax and Inductive Semantics

The grammars for types and terms, together with the type rules are given in Fig. 1. Note that the λ-calculus operators are part of the meta-language. They

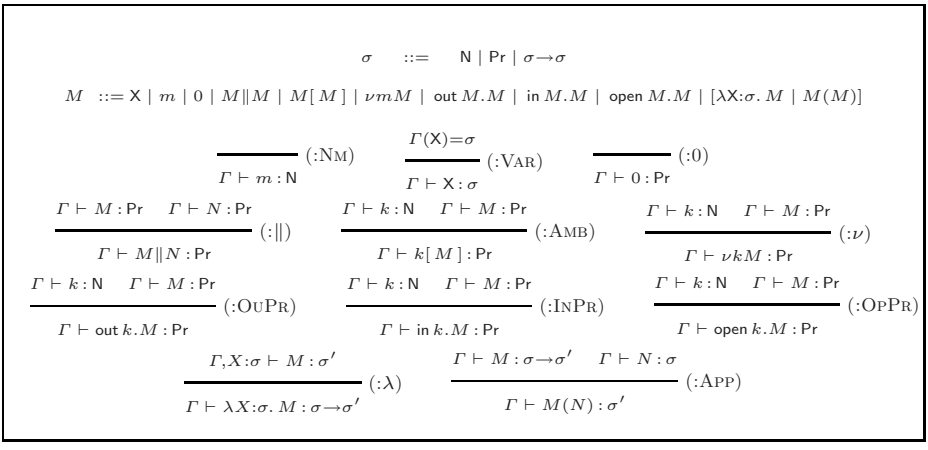


Fig. 1. Types, syntax and type rules

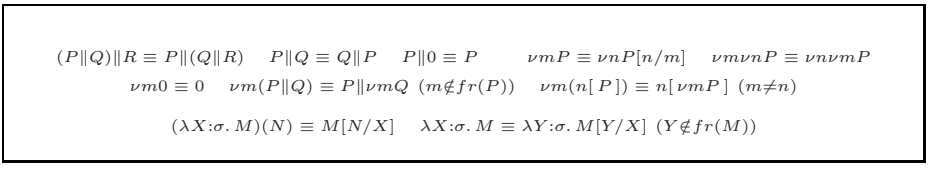


Fig. 2. Structural congruence

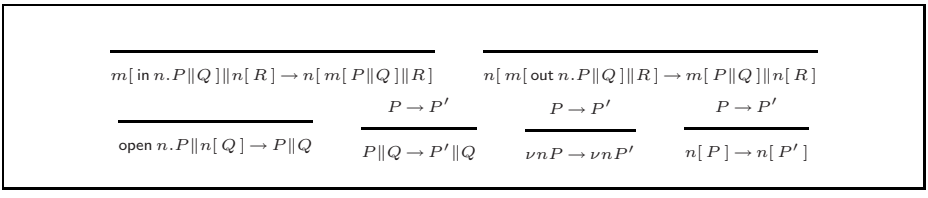


Fig. 3. Reduction semantics, inductively

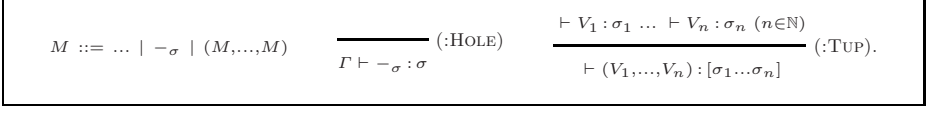


Fig. 4. Interface types

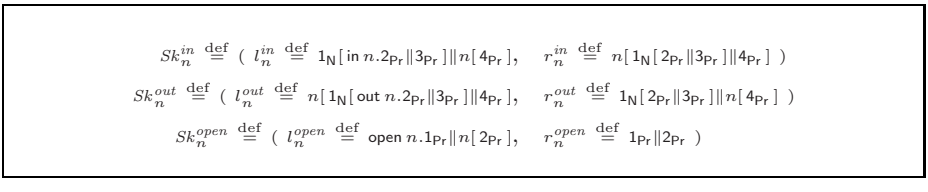


Fig. 5. Skeletons $l_n^{in}, r_n^{in}{:}[\mathsf{N},\mathsf{Pr}^3]{\to}[\mathsf{Pr}]$, $l_n^{out}, r_n^{out}{:}[\mathsf{N},\mathsf{Pr}^3]{\to}[\mathsf{Pr}]$ and $l_n^{open}, r_n^{open}{:}[\mathsf{Pr}^2]{\to}[\mathsf{Pr}]$

are used here solely to define the labelled transition system and should not be considered as a language extension. We assume distinct countable supplies of names (ranged over by n, m) and variables (ranged over by X, Y, x, y). By convention, we will use x, y for variables of type N, X, Y for variables of type Pr, k, l for terms of type N and P, Q, R for closed terms of type Pr. M, N will be used for arbitrary terms of type Pr. A type context Γ is a finite map from variable names to types. We consider only typeable terms. The axioms of structural congruence are given in Fig. 2. It is straightforward to show that any term N structurally congruent to M is typeable iff M is and that they have the same type.

Our transition systems are presented in the structural style. We make one non-standard assumption: we always assume the implicit presence of the rule

$$\frac{P' \equiv P \quad P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{P' \xrightarrow{\alpha} Q'} \text{(StrCng)}.$$

The reduction semantics is given in Fig. 3. It is easy to show that subject reduction holds.

Before we proceed it is worth pointing out that our language does not contain any replication or recursion operator and is thus finite. This however is not a significant restriction though as the crafting of a labelled transition system relies on a study of the immediate interactions of a process with a context. We could easily, but pointlessly, include a replication operator with negligible impact on the LTS rules. Given an LTS $\mathcal{L}$ the only labelled equivalence we shall consider is standard strong bisimilarity $\sim_{\mathcal{L}}$, which is defined as the largest bisimulation on $\mathcal{L}$. Because we wanted to focus on the systematic derivation procedure of LTSs, we have not considered weak equivalences in this paper; our feeling is that the study of weak equivalences examines largely orthogonal issues. We will come back to this issue in the section on future work.

To explain our derivation procedure we will need a general notion of context. Contexts are defined in two stages.

Definition 1 (Precontext). For each type σ, we add *σ-annotated holes* $-_\sigma$ and n-tuples (for any $n \in \mathbb{N}$) to the syntax, together with two additional type rules, given in Fig. 4, where $[\vec{\sigma}]$ is called an *interface type*. A precontext is a typeable term of the form $(V_1,\ldots,V_n)$. Note that each V_i is a closed term in that it does not contain free variables; holes and variables are separate syntactic entities.

Definition 2 (Context). Suppose that a precontext $(\vec{V}):[\vec{\sigma_1}]$ contains m holes. A 1-1 enumeration of the holes with numbers from 1 to m uniquely determines a word $\vec{\sigma_2}$ over types, where $\sigma_{2,i}$ is the type of the ith numbered hole. Syntactically replacing each hole with its number yields a *context* of type $[\vec{\sigma_2}] \to [\vec{\sigma_1}]$. Ordinary terms of type Pr will be identified with contexts of type $[] \to [\mathsf{Pr}]$. Given contexts $f : [\vec{\sigma_1}] \to [\vec{\sigma_2}]$ and $g : [\vec{\sigma_2}] \to [\vec{\sigma_3}]$, there is a context $g \circ f : [\vec{\sigma_1}] \to [\vec{\sigma_3}]$ which is obtained by substitution of the ith component of f for the ith hole of g. This operation may be capturing. Given $f : [\vec{\sigma_1}] \to [\vec{\sigma_2}]$ and $g : [\vec{\sigma_3}] \to [\vec{\sigma_4}]$ let

$f \otimes g : [\vec{\sigma_1}\vec{\sigma_3}] \to [\vec{\sigma_2}\vec{\sigma_4}]$ be the context which puts f and g "side-by-side", where the numbering of all the holes in g are incremented by the length of $\vec{\sigma_1}$. Moreover:

- for any word $\vec{\sigma}=\sigma_1 \ldots \sigma_k$, the *identity* context $\mathrm{id}_{[\vec{\sigma}]} : [\vec{\sigma}] \to [\vec{\sigma}]$ is $(1_{\sigma_1},\ldots,1_{\sigma_k})$;
- if, given $\vec{\sigma}$ and $\vec{\sigma'}$, there exists a permutation $\rho : k \to k$ such that $\forall 1 \leq i \leq k.\ \sigma_{\rho i} = \sigma'_i$, it induces a *permutation context* $\rho : [\vec{\sigma_1}] \to [\vec{\sigma_2}]$ of the form $(\rho 1_{\sigma'_1},\ldots,\rho 1_{\sigma'_k})$;
- a *language context* is a context which does not contain instances of the metalanguage; those of type $[\mathsf{Pr}] \to [\mathsf{Pr}]$ will be denoted by $\mathcal{C}$;

For language contexts we will write $\mathcal{C}[M]$ for $\mathcal{C} \circ M$.

Definition 3 (Barbs). We say that a term P *barbs* on an ambient m, written $P \downarrow_m$, if there is an instance of an ambient m at the "top level". More formally, $P \equiv \nu \vec{n}\,(m[\,Q\,] \parallel R)$ for some $\vec{n}, Q, R$ such that m does not appear in $\vec{n}$.

Definition 4 (Reduction barb congruence). Reduction barb congruence ($\simeq$) is the largest symmetric relation $\mathcal{R}$ such that if $P\ \mathcal{R}\ Q$ then:

(i) If $P \to P'$ then there exists $Q \to Q'$ such that $P'\ \mathcal{R}\ Q'$;
(ii) if $P \downarrow_m$ then $Q \downarrow_m$;
(iii) for all language contexts $\mathcal{C}$ we have that $\mathcal{C}[P]\ \mathcal{R}\ \mathcal{C}[Q]$.

2 Derivation of a Structural LTS

The chief novelty of our paper is the *systematic* presentation of a novel LTS for the ambient calculus. Therefore, before we present the LTS we give an account of our derivation procedure. First, we consider the reduction rules of Fig. 3 as parameterised rules. Fig. 5 contains a rendering of these parameterised reduction rules, referred to as *skeletons*. Essentially, a skeleton is a pair of contexts (l^α_n, r^α_n) which describe the structural changes in passing from l^α_n to r^α_n.

Our LTS is organised into three components: the process-view, in Fig. 6, the context-view, in Fig. 7, and the combined system in Fig. 8. The context-view is the simplest of these and consists of a single "applicative" rule. In the remainder of this section we describe how to analyse the skeletons in order to obtain the process-view rules and how this combines with the context-view.

2.1 Derivation Procedure: Axioms

Treating the skeletons of Fig. 5 as syntax trees, we say that a *match* for Sk^α_n is a subtree with root of type Pr of the LHS l^α_n. More formally, if $l^\alpha_n : [\vec{\sigma}] \to [\mathsf{Pr}]$ is the LHS of a skeleton, a match is a term $\mu^\alpha_n : [\vec{\sigma_1}] \to [\mathsf{Pr}]$ such that there exists $\vec{\sigma_2}$ so that $\vec{\sigma_1}\vec{\sigma_2}$ is a permutation of $\vec{\sigma}$, and there exists a context $\chi : [\mathsf{Pr},\vec{\sigma_2}] \to [\mathsf{Pr}]$ satisfying (†) where $\rho : [\vec{\sigma_1}\vec{\sigma_2}] \to [\vec{\sigma}]$ is the permutation context. A match is said to be *active* if there *does not* exist a context $\chi' : [\mathsf{Pr},\vec{\sigma_2}] \to [\mathsf{Pr}]$ satisfying (‡).

$$\chi \circ (\mu^\alpha_n \otimes \mathrm{id}_{[\vec{\sigma_2}]}) = l^\alpha_n \circ \rho \quad (\dagger) \qquad \chi' \circ (\mu^\alpha_n \otimes \mathrm{id}_{[\vec{\sigma_2}]}) = r^\alpha_n \circ \rho \quad (\ddagger)$$

Intuitively, an active match is a part of the LHS of the skeleton that is modified as a result of the reduction. Clearly any match which has an active match as a subtree is itself active. Of particular interest are those active matches which are locally *minimal* with respect to the subtree relation.

Observation 5. *The minimal active matches for the skeletons in Fig. 5 are:*

- *for* Sk_n^{in}*:* $\mathsf{in}\, n.1_{\mathsf{Pr}}$ *and* $n[\,1_{\mathsf{Pr}}\,]$*;*
- *for* Sk_n^{out}*:* $\mathsf{out}\, n.1_{\mathsf{Pr}}$*;*
- *for* Sk_n^{open}*:* $\mathsf{open}\, n.1_{\mathsf{Pr}}$ *and* $n[\,1_{\mathsf{Pr}}\,]$*.*

The axioms of our process-view LTS are determined by the minimal active matches. Indeed, their LHSs are the *instantiated* minimal active matches: given a minimal active match $\mu_n^\alpha : [\vec{\sigma}] \to [\mathsf{Pr}]$ they are the terms $\mu_n^\alpha \circ \iota$ where $\iota : [] \to [\vec{\sigma}]$. The result is then the RHS of the skeleton instantiated with the parameters ι of the minimal match together with that remaining parameters ι' required by χ:

$$\mu_n^\alpha \circ \iota \xrightarrow{\chi\circ(1_{\mathsf{Pr}}\otimes\iota')} r_n^\alpha \circ \rho \circ (\iota \otimes \iota'). \tag{1}$$

This is clearly an RPO-like observation: the context provides $\chi\circ(1_{\mathsf{Pr}}\otimes\iota')$ and enables a reduction $\chi\circ(1_{\mathsf{Pr}}\otimes\iota')\circ\mu_n^\alpha\circ\iota = \chi\circ(\mu_n^\alpha\otimes\mathrm{id})\circ(\iota\otimes\iota') = l_n^\alpha\circ\rho\circ(\iota\otimes\iota') \to r_n^\alpha\circ\rho\circ(\iota\otimes\iota')$.

Note that each χ is uniquely determined by the particular minimal active match μ_α^n. For this reason in the label of the transition we will use a textual abbreviation $\alpha_i n{\downarrow}\vec{M}$ where $\alpha_i n$ represents the ith minimal active match of Sk_i^α, and $\vec{M}$ the list of the remaining parameters (cf ι' in (1)). Following this procedure, we obtain the following labelled transitions:

$$\mathsf{in}\, n.P \xrightarrow{\mathsf{in}_1\, n{\downarrow}QkR} n[\,k[\,P\|Q\,]\|R\,] \qquad n[\,P\,] \xrightarrow{\mathsf{in}_2\, n{\downarrow}QRk} n[\,k[\,Q\|R\,]\|P\,] \tag{2}$$

$$\mathsf{out}\, n.P \xrightarrow{\mathsf{out}_1\, n{\downarrow}QkR} k[\,P\|Q\,]\|n[\,R\,] \tag{3}$$

$$\mathsf{open}\, n.P \xrightarrow{\mathsf{open}_1\, n{\downarrow}Q} P\|Q \qquad n[\,P\,] \xrightarrow{\mathsf{open}_2\, n{\downarrow}Q} Q\|P \tag{4}$$

The main obstacle in giving a structural derivation of an LTS with labels of the kind given above is that in the results of the above transitions, the distinction between the parts provided by the process and the parts provided by the context is lost. Our solution is to delay the instantiation of the context components. Technically this is done with the use of the meta-syntax – the context contributions are initially replaced with lambda abstracted variables.

The SOS rules are thus naturally divided into three parts - rules for the *process-view* LTS $\mathcal{C}$ for deriving the part of the label to the left of the $\downarrow$ symbol, rules for the *context-view* LTS $\mathcal{A}$ for deriving the remainder of the label, and rules for the *combined* LTS $\mathcal{CA}$ which juxtapose the two contributions to form "complete" labelled transitions. Following this nomenclature, the process-view contribution to the transitions in (2) is

$$\frac{}{\mathsf{in}\, n.P \xrightarrow{\mathsf{in}_1\, n} \lambda \mathsf{X}\mathsf{x}\mathsf{Y}.\ n[\,\mathsf{x}[\,P\|\mathsf{X}\,]\|\mathsf{Y}\,]}\ (\textsc{In}1) \qquad \frac{}{n[\,P\,] \xrightarrow{\mathsf{in}_2\, n} \lambda \mathsf{X}\mathsf{Y}\mathsf{x}.\ n[\,\mathsf{x}[\,\mathsf{X}\|\mathsf{Y}\,]\|P\,]}\ (\textsc{In}2) \tag{5}$$

while the context parts are given by rule (Inst) of Fig. 7. The rule which juxtaposes them is (Cλ) of Fig. 8. We take (In1), (In2) (*cf.* 5), (Ou1) (obtained from (3)), (Op1), (Op2) (obtained from (4)) as provisional axioms for the process-view LTS.

2.2 Derivation Procedure: Structure

Once the axioms are determined, we can attempt to provide the structural rules. There are three kinds:

(i) a *substructural* modification: the added structure takes part in the reduction but the match, and therefore the label, remain unchanged. The structure is added to the appropriate parameter in the RHS. A particular kind of substructural transition used here concerns the situation where the current match is in parallel with a hole of type Pr in the skeleton; *e.g.* the minimal active match of Sk_n^{out}. Using the fact that structural congruence ensures that $(\|, 0)$ is a commutative monoid, introducing a parallel component does not mean that we must expand the match, instead we add the component to the parameter representing the aforementioned hole;

(ii) a *superstructural* modification: the match, and therefore the label, remain unchanged and the added structure does not take part in the reduction; it is added to the result at top level. This situation is common and therefore we will make use of the following abbreviations which deal with lambda abstractions $T{=}\lambda\vec{\mathsf{X}}.P$:

$$T\|Q \stackrel{\text{def}}{=} \lambda\vec{\mathsf{X}}.(T(\vec{\mathsf{X}})\|Q) \qquad \text{and} \qquad \nu m T \stackrel{\text{def}}{=} \lambda\vec{\mathsf{X}}.\nu m T(\vec{\mathsf{X}});$$

(iii) an *observational* modification: the extra structure forces the enlargement of the match as a subtree of its skeleton – here the label itself has to be changed. Once enough structure is added to cover the entire LHS of a skeleton, a τ-labelled transition should be derived. This can occur in two ways, depending on the number of the minimal active matches in the skeleton. These two cases are analysed in the two paragraphs below for the setting of the ambient calculus.

In Sk_n^{out} which has only one minimal active match, the procedure is relatively straightforward. The axiom (Ou) in Fig. 6 is just (Ou1) as described previously, with the numeral omitted. The rule ($\|$Ou) is a substructural modification as described above. The rule (νOu) is a superstructural modification since the ν binder has to first migrate outside, using structural congruence, before the reduction can take place. The side condition enables this emigration. Note that because substitution that is part of β-reduction is capture avoiding, the binder in the result will not bind any names when instantiated by combining with the context-view; the correct behaviour. The rule (OuAmb) is an observational modification, here the structure (the ambient n) forces us to expand the match within the skeleton, meaning that we can now instantiate the first two parameters. The rule ($\|$OuAmb) is substructural while (νOuAmb) is superstructural. Finally, (OuTau) is an observational modification which completes the skeleton, meaning that we derive a τ-labelled transition.

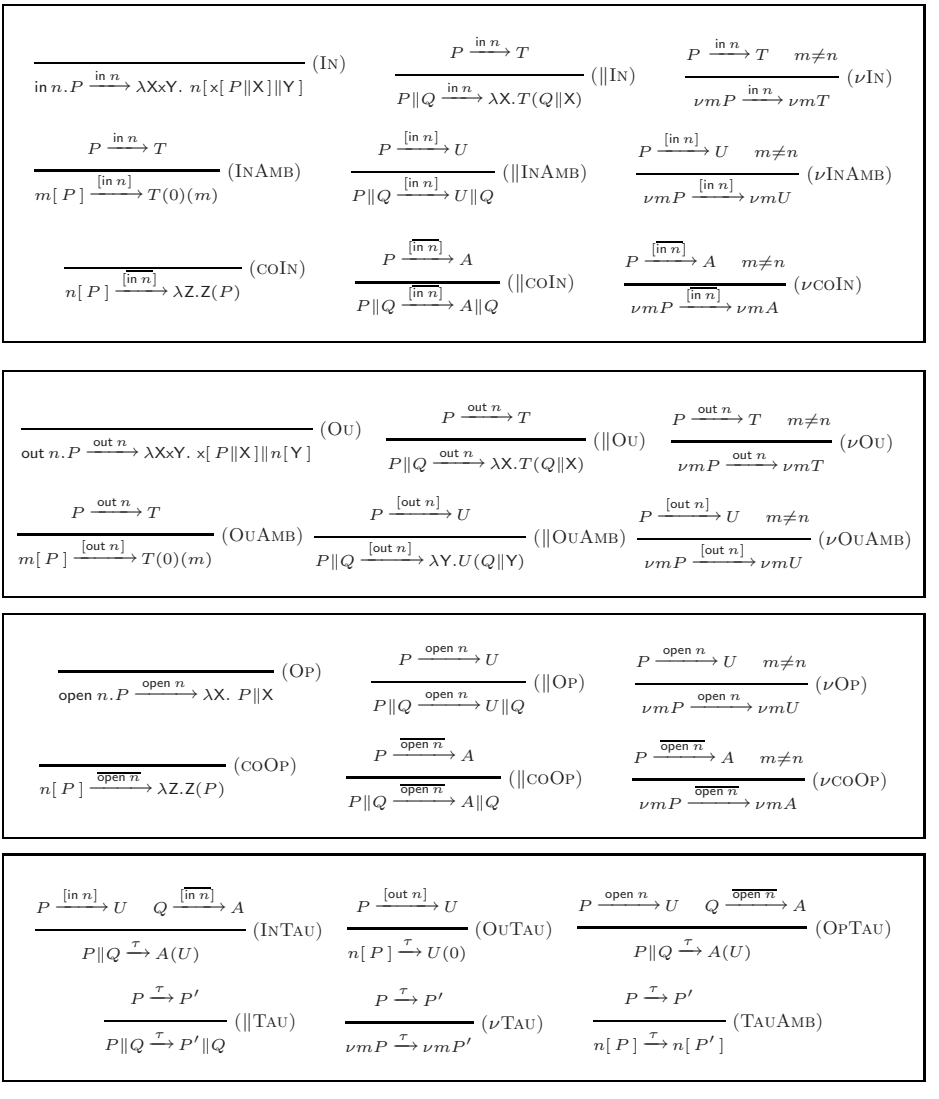


Fig. 6. Process-view fragment ($\mathcal{C}$). By convention $T : \mathsf{Pr} \to \mathsf{N} \to \mathsf{Pr} \to \mathsf{Pr}$, $U : \mathsf{Pr} \to \mathsf{Pr}$, $A : (\mathsf{Pr} \to \mathsf{Pr}) \to \mathsf{Pr}$.

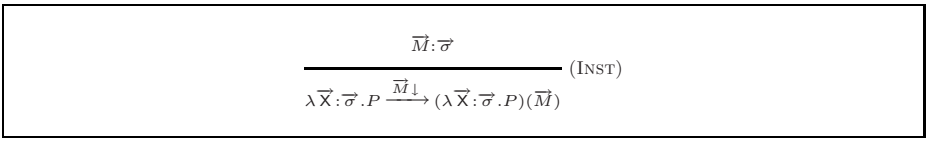


Fig. 7. Context-view fragment ($\mathcal{A}$)

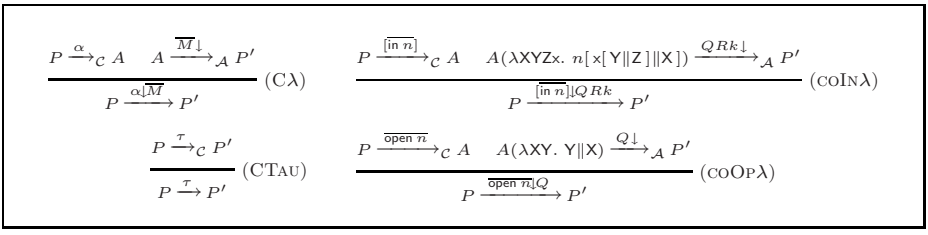


Fig. 8. Combined system of complete actions ($\mathcal{CA}$)

Skeletons with two (or more) minimal active matches lead to a more complicated situation. Indeed, consider the two minimal active matches of Sk_n^{in} and the two corresponding provisional axioms given in (5). Starting with either one, structure can be added extending the match. Indeed, consider (In) of Fig. 6 which is obtained from (In1) of (5) by omitting the numeral. The rule ($\|$In) is substructural and (νIn) superstructural. The rule (InAmb) is observational and extends the minimal match with the surrounding ambient. No further extension of the match is possible without including the contribution of the second minimal active match. The structural approach requires the combination of observations of the two matches in order to cover the entire LHS of the skeleton and derive a τ. However, in our two provisional axioms (In1), (In2) we have included the RHS of the skeleton in result of the transitions, and it is not obvious how to "merge" the two, collecting the appropriate parameters. Our solution is to use *co-actions*, borrowing continuation-passing style. Indeed, we discard (In2) and instead use the axiom (coIn) of Fig. 6. The idea is that instead of using the actual skeleton in the result, we use an abstract skeleton and apply that to the parameter (of the minimal active match). Merging actions and co-actions is now easy as the abstract skeleton can be replaced by the the actual skeleton provided by the action. Superstructural rules ($\|$coOp) and (νcoOp) are straightforward and we are able to use (InTau) to collect the parameters to the RHS of the skeleton using a simple application. A similar approach is used to deal with the *open* reduction.

The use of co-actions gives one final complication. Because the result of a co-action transition does not have the shape which would result from using the RHS of the skeleton, we cannot simply use the the combination of (Inst) of Fig. 7 and (Cλ) of Fig. 8. Instead, we use the rules (coInλ) and (coOpλ) which insist that the context provided by the environment conforms to the skeleton.

It is worth clarifying as to what extent the procedure, as described above, is systematic. As we have explained, we have chosen to include the RHS in the right hand side of (In1), resulting in (In). Differently, and in seemingly ad-hoc fashion, we have not done this for (In2), using instead a co-action (CoIn). A more systematic presentation would consist in using the co-action style for *all* the labels. Following this approach, the actual skeleton would never actually be instantiated in the RHS of the process-view transitions. The main price for this is that the rule (Inst) needs to be replaced with specific rules for each co-action, in the spirit of (CoInλ) and (CoOpλ) of Fig. 8. Such an "all-co-action" SOS rule set

would derive the same LTS as the rule set presented in this paper. We believe that this approach could be mechanised. We have chosen to present the rules as in Fig. 6 because we believe that they are easier to understand, and more importantly, they correspond more closely to rules in previously published SOS rule sets for the ambient calculus (*cf.* Section 4).

The following lemma provides a sanity-check for our LTS which ensures that the transitions obtained from our structural rules are justified by a reduction in a context; the point with which we started our discussion in (1) on page 467.

Lemma 6. *If* $P \xrightarrow{\alpha\lfloor\vec{M}} _{\mathcal{CA}} P'$, *then there exists a context* χ_α *such that* $\chi_\alpha \circ (1_{\mathsf{Pr}}, \vec{M}) \circ P \to P'$. *We list the corresponding* χ_α*s below:*

$$\chi_{\mathsf{in}\ n} \stackrel{\text{def}}{=} 3_{\mathsf{N}}[\,1_{\mathsf{Pr}} \| 2_{\mathsf{Pr}}\,] \| n[\,4_{\mathsf{Pr}}\,] \qquad \chi_{[\mathsf{in}\ n]}, \chi_{\mathsf{open}\ n} \stackrel{\text{def}}{=} 1_{\mathsf{Pr}} \| n[\,2_{\mathsf{Pr}}\,] \qquad \chi_{\overline{[\mathsf{in}\ n]}} \stackrel{\text{def}}{=} 4_{\mathsf{N}}[\,\mathsf{in}\ n.2_{\mathsf{Pr}} \| 3_{\mathsf{Pr}}\,] \| 1_{\mathsf{Pr}}$$

$$\chi_{\overline{\mathsf{open}\ n}} \stackrel{\text{def}}{=} \mathsf{open}\ n.2_{\mathsf{Pr}} \| 1_{\mathsf{Pr}} \qquad \chi_{\mathsf{out}\ n} \stackrel{\text{def}}{=} n[\,3_{\mathsf{N}}[\,1_{\mathsf{Pr}} \| 2_{\mathsf{Pr}}\,] \| 4_{\mathsf{Pr}}\,] \qquad \chi_{[\mathsf{out}\ n]} \stackrel{\text{def}}{=} n[\,1_{\mathsf{Pr}} \| 2_{\mathsf{Pr}}\,] \qquad \chi_\tau \stackrel{\text{def}}{=} 1_{\mathsf{Pr}}$$

3 Soundness and Completeness

Having presented our new labelled transition system we must demonstrate that it is fit for purpose. Specifically, the τ labelled transitions must characterise the reductions. Moreover, we also require that bisimulation equivalence is sound for reduction barb congruence. The fact that $\xrightarrow{\tau} \subseteq \to$ is implied already by the conclusion of Lemma 6. The converse follows by a straightforward inductive analysis of the structural forms of processes which may generate τ transitions.

Proposition 7 (Tau and Reduction). $P \xrightarrow{\tau} P'$ *iff* $P \to P'$. □

The chief property that needs to be established for the latter requirement (the soundness of $\sim_{\mathcal{CA}}$ with respect to $\simeq$) is congruence of bisimilarity with respect to language contexts. As a consequence of the fact that the LTS follows from the construction outlined in §2, this is straightforward to establish. The case of observational modifications which combine two separate derivations is the most interesting; here this concerns the rules (InTau) and (OpTau). Because the combination occurs via the $\|$ operator, these rules are considered within a sub-case of the proof that bisimilarity is a congruence with respect to $1_{\mathsf{Pr}} \parallel P$ contexts. The argument is roughly the following: the target of the derived τ-labelled transition, an application of the targets of two process-view transitions, can also be obtained by completing one of the transitions with the result of the other. The inductive hypothesis can then used in order to match this complete transition, resulting in a bisimilar state, which can then be again deconstructed.

Proposition 8 (Congruence). *If* $P \sim_{\mathcal{CA}} Q$ *then* $\mathcal{C}[P] \sim_{\mathcal{CA}} \mathcal{C}[Q]$ *for all language contexts* $\mathcal{C}$. □

Theorem 9 (Soundness). $P \sim_{\mathcal{CA}} Q$ *implies* $P \simeq Q$. □

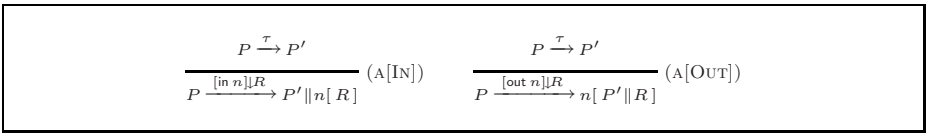


Fig. 9. Honda-Tokoro rules $\mathcal{HT}$ for unobservable actions

With soundness of bisimilarity established, it is a natural question as to whether the converse property of completeness holds. This is complicated by the issue of observability of actions. As encapsulated by the statement of Lemma 6, the labels of our LTS have corresponding underlying context-triggered reductions. Completeness relies on the converse relationship; a context-triggered reduction (or series of reductions and barb-observations) implying the existence of a transition.

Completeness needs to be checked manually; our systematic derivation technique as outlined in §2 does not guarantee that it holds. To prove it, one needs to show that each kind of label has a context which characterises it. This is a stronger requirement then that of Lemma 6 which exhibits a relationship between contexts and labels in one direction only: every labelled transition has a corresponding context in which there is a reduction to the right hand side. However, a reduction in this context does not necessarily imply the existence of the labelled transition. In order to do this, contexts have to contain more information.

It is unclear whether the LTS is complete with respect to reduction barb congruence in our finite language. Simply adding replication to the language does result in a language for which the LTS is not complete. Indeed, in the full ambient calculus, an ambient's ability to migrate is unobservable. This fact has been observed in [18] and a suitable adaptation of the definition of bisimulation is given to account for this. For aesthetic reasons we prefer to use ordinary bisimulation and thus use, a suitable modification of the Honda-Tokoro [15] style rules for strong equivalences instead. We have began a more general investigation of such rules in [25]. Interestingly, here they are needed for $[\mathsf{in}\, n]$ and $[\mathsf{out}\, n]$ transitions only (*cf.* Fig. 9) and account for the following situation: the context provides the appropriate χ (*cf.* Lemma 6) but the process does not make use of it, thus χ is retained in the result. The rules are added to the combining rules of Fig. 8. Bisimilarity $\sim_{(\mathcal{C}+\mathcal{HT})\mathcal{A}}$ on the obtained LTS remains sound for contextual equivalence.

As an example of the necessity of the $\mathcal{HT}$ rules for completeness, consider:

$$T_1 \stackrel{\text{def}}{=} !n[\,0\,] \parallel \nu k(\, k[\,\mathsf{in}\, n.0\,]\,) \qquad \text{and} \qquad T_2 \stackrel{\text{def}}{=} !n[\,0\,] \parallel \tau$$

where $\tau \stackrel{\text{def}}{=} \nu m(\mathsf{open}\, m.0 \parallel m[\,0\,])$. Processes T_1 and T_2 are reduction barb congruent. It is not difficult to check this directly using the fact that $\nu k\, k[\,0\,] \sim_{\mathcal{CA}} 0$.

Nevertheless $T_1 \not\sim_{\mathcal{CA}} T_2$ because the T_1 can do a $[\mathsf{in}\, n] \downarrow R$ transition which cannot be matched by T_2. Instead, it *does* hold that $T_1 \sim_{(\mathcal{C}+\mathcal{HT})\mathcal{A}} T_2$:

$$T_1 \xrightarrow{[\mathsf{in}\, n]\!]R} !n[\,0\,] \parallel \nu k(\, n[\, k[\,0\,] \parallel R\,]\,) \text{ is matched by } T_2 \xrightarrow{[\mathsf{in}\, n]\!]R} !n[\,0\,] \parallel n[\, R\,].$$

Concerning the remaining possible labels not considered by $\mathcal{HT}$-rules, we need to show that each complete labelled transition can be characterised by a predicate which is stable under reduction barbed congruence. For example, to characterise the transition labelled with $\mathsf{open}\, n \downarrow R$ we use the context $\xi \stackrel{\text{def}}{=} 1_{\mathsf{Pr}} \parallel n[\, i[\, 0\,] \parallel \mathsf{open}\, i.R\,]$ (with i fresh) and then show that $P \xrightarrow{\mathsf{open}\, n \downarrow R} P'$ iff there exists P'' such that $\xi[P] \rightarrow P''$ with $P'' \downarrow_i$ and $P'' \rightarrow P'$ with $P' \not\downarrow_i$.

Theorem 10 (Completeness). *$P \simeq Q$ implies $P \sim_{(\mathcal{C}+\mathcal{HT})\mathcal{A}} Q$.* □

4 Conclusions, Related and Future Work

The introduction of the ambient calculus in [7] has spawned an enormous amount of research on the topic regarding variants of the calculus (*e.g.* [3, 10, 11]), type systems (*e.g.* [4, 6, 19]) and implementation details (*e.g.* [13, 22]). However, there has been relatively little work on labelled characterisations. An early attempt by Cardelli and Gordon [5] was abandoned in favour of a simpler approach in [8]. Interestingly, the structural rules and use of abstractions in the meta-language was already present in [5] where the authors seemed to encounter difficulty lay in relating their structural labels to contexts. This was particularly true for co-actions. The approach that we take in this paper resolves this issue.

Subsequent to [5, 8], Merro and Zappa-Nardelli [18] designed an LTS and established a full abstraction result using a form of context bisimilarity. Their paper is ostensibly the approach most closely related to ours in terms of results but the emphasis in our research is on a *systematic* derivation of the LTS model to achieve a similar result. We were fortunate in having had the model in [18] to use as a comparison and sanity check for our own semantics.

We hope that the benefits of our approach will become clear once one has compared the two LTS models: Merro and Zappa-Nardelli produced an LTS which built on the initial attempts by Cardelli and Gordon [5] (which already contained a reasonable account of the structural transitions towards an inductive definition of the τ-reduction relation) by analysing the contextual interactions provided by an arbitrary environment. Doing this necessitated a restriction to *system* level ambients – that is, ambients which were all boxed at top level – and a use of a piece of meta-syntax $\circ$ to allow arbitrary environmental processes to be re-inserted into terms. The latter of these requirements resurfaces in our work through the use of the λ-calculus meta-language but the former, the restriction to systems, is avoided by providing context-oriented structural transitions in the LTS $\mathcal{C}$. The effect of this is that *all* of our (completed) labelled transitions are suitable for use in the definition of bisimulation as opposed to only the class of env-actions in [18]. Notice, for example, that our base rules (In) and (Ou) of Fig. 6 retain the structure of the interacting context and term. This structure is carried in the rules (InAmb) and (OuAmb) whereas Merro and Zappa-Nardelli's related rules, (Enter Shh) and (Exit Shh), in [18] serve primarily to recover this necessary structure. Our treatment of co-actions, in rules (coInλ) and (coOpenλ) of Fig. 8, by completing them with skeletal structure as well as missing parameters,

is mirrored in the rules (Co-Enter) and (Open) of [18] although the restriction to systems complicates the latter of those. The remaining difference lies in the use of the name enclosing the migrating ambient in the (Enter) and (Exit) rules. They are included as part of the label in [18] and therefore reflect a slightly finer analysis of observability in ambients. However, rules (Enter Shh) and (Exit Shh) are then necessary because this name is not always observable. Our equivalent rules (InAmb) and (OutAmb) do not record the name of the enclosing ambient in the label because this information is not determined by the context and the visibility of this name to be discovered by context parameter processes instead. Unlike [18] we deal with the unobservability of $[\mathsf{in}\, n]$ and $[\mathsf{out}\, n]$ actions using Honda Tokoro style [15] rules in Fig. 9 rather than adopting a non-standard definition of bisimulation in the style of [1]. In conclusion, our derived LTS is pleasingly similar to, and, we believe, conceptually cleaner than its counterpart in [18] which represents the state of the art for this language to date.

In addition to the work mentioned above there have been a number of LTS models for variants of the ambient calculus [3, 10, 11, 12]. These models all use a variant of the language for which the contextual observations of co-actions are much clearer than in the pure ambient model and therefore the co-action labelled transitions are more easily defined. It will be interesting to see how our methodology fares when applied to these variants.

Finally, it is interesting to note that Sewell has already considered applying his contexts-as-labels approach [26] to the ambient calculus. We note that this work already suggests using (non-inductive versions of) our rules (In), (Out), and (Open). Similarly, Jensen and Milner [16] use the context-as-labels approach to provide a derived LTS for the ambient calculus via an encoding to bigraphs. This LTS is also non-inductive and the lack of a detailed analysis of the resulting RPOs in [16] makes it difficult for us to find any striking similarities with our SOS rule-set and LTS.

In this paper and in [24] we have considered strong bisimilarity. It is interesting to observe that because Proposition 7 holds and because our bisimulation equivalence is defined over complete actions $\mathcal{CA}$, in principle it should be possible to smoothly lift our soundness and completeness results to weak bisimilarity. Notably, for weak transitions

$$P \xrightarrow{\tau}_{\mathcal{CA}} \cdots \xrightarrow{\alpha}_{\mathcal{CA}} \cdots \xrightarrow{\tau}_{\mathcal{CA}} P'$$

we will only ever need to decompose the *strong* α transition in to its Process and Context views. In particular, to characterise the weak equivalences, it is **not** the case that we will need to consider weak transitions from the $\mathcal{C}$ and $\mathcal{A}$ transitions systems separately. The difficulties which may arise in the weak case lie in providing contexts which witness weak transitions for the proof of completeness. We do not anticipate problems here but we have not yet checked the details of this.

The separation of process- and context-views in our approach means that our bisimulation equivalences are context bisimulations. This is due to the completion of labels by considering arbitrary context processes. As shown in [24], it is

sometimes possible to exploit this separation in order to refine the context-view so that only certain archetypal context processes need be supplied. It would be interesting to attempt to design an analogous refinement for ambients and we believe that our LTS serves as a good basis from which to do this.

Having experimented on the π-calculus [24] and the ambient calculus, we now intend to develop our method for deriving transition systems in a general setting and establish soundness and completeness results for a wide range of calculi.

Acknowledgment. We would like to thank the anonymous referees for their useful comments which have helped to improve the presentation of the paper.

References

1. Amadio, R., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. Theor. Comput. Sci. 195(2), 291–324 (1998)
2. Bergstra, J., Klop, J.: Algebra of communicating processes with abstraction. Theor. Comput. Sci. 37(1), 77–121 (1985)
3. Bugliesi, M., Crafa, S., Merro, M., Sassone, V.: Communication interference in mobile boxed ambients. Information and Computation 205, 1235–1273 (2007)
4. Cardelli, L., Ghelli, G., Gordon, A.: Mobility types for mobile ambients. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 230–239. Springer, Heidelberg (1999)
5. Cardelli, L., Gordon, A.: A commitment relation for the ambient calculus (unpublished notes, 1996)
6. Cardelli, L., Gordon, A.: Types for mobile ambients. In: Proc. POPL, pp. 79–92. ACM Press, New York (1999)
7. Cardelli, L., Gordon, A.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (2000)
8. Cardelli, L., Gordon, A.: Equational properties of mobile ambients. Math. Struct. Comput. Sci. 13(3), 371–408 (2003)
9. Engberg, U., Nielsen, M.: A calculus of communicating systems with label passing. Technical Report DAIMI PB-208, University of Aarhus (May 1986)
10. Fu, Y.: Fair ambients. Acta Inf. 43(8), 535–594 (2007)
11. Garralda, P., Bonelli, E., Compagnoni, A., Dezani-Ciancaglini, M.: Boxed Ambients with Communication Interfaces. MSCS 17, 1–59 (2007)
12. Hennessy, M., Merro, M.: Bisimulation congruences in safe ambients. ACM Transactions on Programming Languages and Systems 28(2), 290–330 (2006)
13. Hirschkoff, D., Pous, D., Sangiorgi, D.: An efficient abstract machine for safe ambients. J. Logic and Algebraic Programming 71, 114–149 (2007)
14. Hoare, C.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
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. Jensen, O., Milner, R.: Bigraphs and mobile processes. Technical Report 570, Computer Laboratory. University of Cambridge (2003)
17. Leifer, J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000)

18. Merro, M., Nardelli, F.Z.: Behavioural theory for mobile ambients. J. ACM 52(6), 961–1023 (2005)
19. Merro, M., Sassone, V.: Typing and subtyping mobility in boxed ambients. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 304–320. Springer, Heidelberg (2002)
20. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
21. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, II. Inf. Comput. 100(1), 41–77 (1992)
22. Phillips, A.: Specifying and Implementing Secure Mobile Applications in the Channel Ambient System. PhD thesis, Imperial College London (April 2006)
23. Plotkin, G.: A structural approach to operational semantics. Technical Report FN-19, DAIMI, Computer Science Department, Aarhus University (1981)
24. Rathke, J., Sobociński, P.: Deconstructing behavioural theories of mobility. In: Proc. IFIP TCS 5th International Conference on Theoretical Computer Science. Springer Science and Business Media (to appear, 2008)
25. Rathke, J., Sobociński, P.: Making the unobservable unobservable. In: Proc. ICE 2008. ENTCS. Elsevier, Amsterdam (to appear, 2008)
26. Sewell, P.: From rewrite rules to bisimulation congruences. Theor. Comput. Sci. 274(1-2), 183–230 (2002); Extended version of Concur 1998 conference paper

Termination Problems in Chemical Kinetics

Gianluigi Zavattaro[1] and Luca Cardelli[2]

[1] Dip. Scienze dell'Informazione, Università di Bologna, Italy
[2] Microsoft Research, Cambridge, UK

Abstract. We consider nondeterministic and probabilistic termination problems in a process algebra that is equivalent to basic chemistry. We show that the existence of a terminating computation is decidable, but that termination with any probability strictly greater than zero is undecidable. Moreover, we show that the fairness intrinsic in stochastic computations implies that termination of all computation paths is undecidable, while it is decidable in a nondeterministic framework.

1 Introduction

We investigate the question of whether basic chemical kinetics (kinetics of unary and binary chemical reactions), formulated as a process algebra, is capable of general computation. In particular, we investigate nondeterministic and probabilistic termination problems in the Chemical Ground Form (CGF): a process algebra recently proposed for the compositional description of chemical systems, and proved to be both stochastically and continuously equivalent to chemical kinetics (see [2] for the formal proof of equivalence between CGF and chemical kinetics). The answers to those termination problems reveal a surprisingly rich picture of what is decidable and undecidable in basic chemistry.

We consider three variants of the termination problem: *existential*, *universal*, and *probabilistic* termination. By existential termination we mean the existence of a terminating computation, by universal termination we mean that all possible computations terminate (in a probabilistic setting, by possible computation we mean that the computation has probability > 0), by probabilistic termination we mean that with probability strictly greater than a given ϵ, with $0 < \epsilon < 1$, a terminating computation is executed. We prove that, in the stochastic semantics of CGF, existential termination is decidable, while both probabilistic and universal termination are undecidable. In contrast, in a nondeterministic interpretation of the CGF that abstracts from reaction rates, both existential and universal termination are decidable. This means that: (a) chemical kinetics is not Turing complete, (b) chemical kinetics is Turing complete up to any degree of precision, (c) existential termination is equally hard (decidable) in stochastic and nondeterministic systems, (d) universal termination is harder (undecidable) in stochastic systems than in nondeterministic systems, (e) the fairness implicit in stochastic computations makes checking universal termination undecidable.

In recent work, Soloveichik et al. [8], prove the non-Turing completeness of Stochastic Chemical Reaction Networks (which are equivalent to the CGF [2])

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 477–491, 2008.

by reduction to the decidability of chemical state coverability, which they call reachability. We prove more strongly that exact chemical state reachability is also decidable, as well as that existential termination and boundedness are decidable. (All these argument are based on decidability results in Petri Nets.) The same authors also prove the possibility of approximating RAM and Turing Machine computations up to an arbitrarily small error ϵ. Their encodings allow them to prove the undecidability of probabilistic coverability. We prove the undecidability of probabilistic termination, probabilistic reachability, probabilistic boundedness, and of universal termination. There are technical differences in our RAM encodings that guarantee the stronger results. For example, terminating computations are still terminating in our encoding of RAMs, while in [8] a "clock" process keeps running even after termination of the main computation.

2 Chemical Ground Form

In the CGF each species has an associated definition describing the possible actions for the molecules of that species. Each action $\pi_{(r)}$ has an associated stochastic rate r (a positive real number) which quantifies the expected execution time for the action π. Action $\tau_{(r)}$ indicates the possibility for a molecule to be engaged in a unary reaction. For instance, the definition $A = \tau_{(r)}; (B|C)$ says that one molecule of species A can be engaged in a unary reaction that produces two molecules, one of species B and one of species C (the operator "|" is borrowed from process algebras such as CCS [6], where it represents parallel composition, and corresponds here to the chemical "+"). Binary reactions have two reactants. The two reactants perform two complementary actions $?a_{(r)}$ and $!a_{(r)}$, where a is a name used to identify the reaction; both the name a and the rate r must match for the reaction to be enabled. For instance, given the definitions $A = ?a_{(r)}; C$ and $B = !a_{(r)}; D$, we have that two molecules of species A and B can be engaged in a binary reaction that produces two molecules, one of species C and one of species D. If the molecules of one species can be engaged in several reactions, then the corresponding definition admits a choice among several actions. The syntax of choice is as follows: $A = \tau_{(r)}; B \oplus ?a_{(r')}; C$, meaning that molecules of species A can be engaged in either a unary reaction, or in a binary reaction with another molecule able to execute the complementary action $!a_{(r')}$.

Definition 1 (Chemical Ground Form (CGF)). *Consider the following denumerable sets:* Species *ranged over by variables* $X, Y, \cdots$, Channels *ranged over by* $a, b, \cdots$, *Moreover, let* $r, s, \cdots$ *be rates (i.e. positive real numbers).*

The syntax of CGF is as follows (where the big $\big|$ *separates syntactic alternatives while the small* $|$ *denotes parallel composition):*

$$
\begin{array}{rcll}
E & ::= & \mathbf{0} \;\big|\; X = M, E & \textit{Reagents} \\
M & ::= & \mathbf{0} \;\big|\; \pi; P \oplus M & \textit{Molecule} \\
P & ::= & \mathbf{0} \;\big|\; X|P & \textit{Solution} \\
\pi & ::= & \tau_{(r)} \;|\; ?a_{(r)} \;|\; !a_{(r)} & \textit{Internal, Input, Output prefix} \\
CGF & ::= & (E, P) & \textit{Reagents and initial Solution}
\end{array}
$$

Given a CGF (E, P), we assume that all variables in P occur also in E. Moreover, for every variable X in E, there is exactly one definition $X = M$ in E.

In the following, trailing $\mathbf{0}$ are left implicit, and we use $|$ also as an operator over the syntax: if P and P' are $\mathbf{0}$-terminated lists of variables, according to the syntax above, then $P|P'$ means appending the two lists into a single $\mathbf{0}$-terminated list. Thus, if P is a solution, then $\mathbf{0}|P$, $P|\mathbf{0}$, and P are syntactically equal. The solution composed of k instances of X is denoted with $\prod_k X$.

We consider the discrete state semantics for the CGF defined in [2] in terms of Continuous Time Markov Chains (CTMCs). The states of the CTMCs are solutions in normal form denoted with $P^\dagger$: for a solution P, we indicate with $P^\dagger$ the normalized form of P where the variables are sorted in lexicographical order (with $\mathbf{0}$ at the end), possibly with repetitions. The CTMC associated to a chemical ground form is obtained in two steps: we first define the Labeled Transition Graph (LTG) of a chemical ground form, then we show how to extract a CTMC from the labeled transition graph.

We use the following notation. Let $E.X$ be the molecule defined by X in E, and $M.i$ be the i-th summand in a molecule of the form $M = \pi_1; P_1 \oplus \cdots \oplus \pi_n; P_n$. Given a solution in normal form $P^\dagger$, with $P^\dagger.m$ we denote the m-th variable in $P^\dagger$, with $P^\dagger \setminus (m_1, \cdots, m_n)$ we denote the solution obtained by removing from $P^\dagger$ the m_i-th molecule for each $i \in \{1, \cdots, n\}$.

A *Labeled Transition Graph* (LTG) is a set of quadruples $\langle l : S^\dagger \xrightarrow{r} T^\dagger \rangle$ where the transition labels l are either of the form $\{m.X.i\}$ or $\{m.X.i, n.Y.j\}$, where m, n, i, j are positive integers, X, Y are species names, $m.X.i$ are ordered triples and $\{\cdots, \cdots\}$ are unordered pairs.

Definition 2 (Labeled Transition Graph (LTG) of a Chemical Ground Form). *Given the Chemical Ground Form (E, P), we define $Next(E, P)$ as the set containing the following kinds of labeled transitions:*

- $\langle \{m.X.i\} : P^\dagger \xrightarrow{r} T^\dagger \rangle$ *such that* $P^\dagger.m = X$ *and* $E.X.i = \tau_{(r)}; Q$ *and* $T = (P^\dagger \setminus m)|Q$;
- $\langle \{m.X.i, n.Y.j\} : P^\dagger \xrightarrow{r} T^\dagger \rangle$ *such that* $P^\dagger.m = X$ *and* $P^\dagger.n = Y$ *and* $m \neq n$ *and* $E.X.i = ?a_{(r)}; Q$ *and* $E.Y.j = !a_{(r)}; R$ *and* $T = (P^\dagger \setminus m, n)|Q|R$.

The Labeled Transition Graph of (E, P) is defined as follows:

$$LTG(E, P) = \bigcup_n \Psi_n$$
$$\text{where } \Psi_0 = Next(E, P) \text{ and } \Psi_{n+1} = \bigcup \{Next(E, Q) \mid Q \text{ is a state of } \Psi_n\}$$

We now define how to extract from an LTG the corresponding CTMC.

Definition 3 (Continuous Time Markov Chain associated to an LTG). *If Ψ is an LTG, then $|\Psi|$ is the associated CTMC, defined as the set of the triples $P \overset{r}{\mapsto} Q$ with $P \neq Q$, obtained by summing the rates of all the transitions in Ψ that have the same source and target state: $|\Psi| = \{P \overset{r}{\mapsto} Q \text{ s.t. } \exists \langle l : P \xrightarrow{r'} Q \rangle \in \Psi \text{ with } P \neq Q, \text{ and } r = \sum r_i \text{ s.t. } \langle l_i : P \xrightarrow{r_i} Q \rangle \in \Psi\}$.*

It is worth noting that two solutions $Q^\dagger$ and $R^\dagger$ are connected by a transition in $LTG(E,P)$ if and only if they are connected by a transition in $|LTG(E,P)|$. In fact, the transitions of the latter are achieved by collapsing into one transition those transitions of the former that share the same source and target solutions. The rate of the new transition is the sum of the rates of the collapsed transitions.

Given a CGF (E,P), a *computation* is a sequence of transitions in the CTMC $|LTG(E,P)|$ starting with a transition with source solution $P^\dagger$, and such that the target solution of one transition coincides with the source state of the next transition. We say that a solution Q is *reachable* in (E,P) if there exists a computation with $Q^\dagger$ as the target solution of the last transition. A solution Q is *terminated* in $|LTG(E,P)|$ if $Q^\dagger$ has no outgoing transitions.

The CTMC semantics of CGF defines a probabilistic interpretation of the behavior of a CGF (E,P): given any solution $T^\dagger$ of $|LTG(E,P)|$, if it has n outgoing transitions labeled with $r_1, \cdots, r_n$, the probability that the j-th transition is taken is $r_j/(\sum_i r_i)$. Thus, we can associate probability measures (we consider the standard probability measure for Markov chains —see e.g. [5]) to computations in $|LTG(E,P)|$. We use this technique to define the three variants of the termination problem we consider in this paper.

Definition 4 (Existential, universal and probabilistic termination). *Consider a CGF (E,P) and its CTMC $|LTG(E,P)|$. Let p be the probability measure associated to the computations in $|LTG(E,P)|$ leading to a terminated solution. We say that (E,P)* existentially terminates *if $p > 0$, (E,P)* universally terminates *if $p = 1$, (E,P)* probabilistically terminates *with probability higher than ϵ (for $0 < \epsilon < 1$) if $p > \epsilon$.*

We will consider also probabilistic variants of other properties. Consider a CGF (E,P), its CTMC $|LTG(E,P)|$, and a real number ϵ such that $0 \leq \epsilon < 1$. We say that a solution Q is *ϵ-reachable* if the probability measure of the computations in $|LTG(E,P)|$ leading to $Q^\dagger$ is $> \epsilon$. We say that (E,P) is *ϵ-bound* if the set of ϵ-reachable solutions is finite. We say that (E,P) is *ϵ-terminating* if the probability measure of the computations in $|LTG(E,P)|$ leading to a terminated solution is $> \epsilon$. We say that (E,P) is *ϵ-diverging* if the probability measure of the infinite computations in $|LTG(E,P)|$ is $> \epsilon$.

It is worth noting that, in a probabilistic setting, existential termination coincides with 0-termination, universal termination with the negation of 0-divergence, and probabilistic termination with ϵ-termination for $\epsilon > 0$.

3 Decidability Results

In this section we resort to a Place/Transition Petri net (P/T net) semantics for CGF, that can be interpreted as a purely nondeterministic semantics of CGF that abstracts away from the stochastic rates. In this purely nondeterministic framework several properties are decidable. In fact, in P/T nets, properties such as *reachability* (the existence of a computation leading to a given state), *boundedness* (the finiteness of the set of reachable states), *termination* (reachability of

a deadlocked state), and *divergence* (the existence of an infinite computation) are decidable (see [4] for a survey on decidable properties for Petri Nets).

Definition 5 (Place/Transition Net). *A P/T net is a tuple* $N = (S, T)$ *where* S *is the set of* places, $\mathcal{M}_{fin}(S)$ *is the set of the finite multisets over* S *(each of which is called a* marking*) and* $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ *is the set of* transitions. *A transition* (c, p) *is written* $c \Rightarrow p$. *The marking* c, *represents the tokens to be "consumed"; the marking* p *represents the tokens to be "produced". A transition* $c \Rightarrow p$ *is* enabled *at a marking* m *if* $c \subseteq m$. *The execution of the transition produces the marking* $m' = (m \setminus c) \oplus p$ *(where* $\setminus$ *and* $\oplus$ *are the difference and the union operators on multisets). This is written as* $m[\rangle m'$. *A* dead marking *is a marking in which no transition is enabled. A* marked *P/T net is a tuple* $N(m_0) = (S, T, m_0)$, *where* (S, T) *is a P/T net and* m_0 *is the* initial marking. *A computation in* $N(m_0)$ *leading to the marking* m *is a sequence* $m_0[\rangle m_1[\rangle m_2 \cdots m_n[\rangle m$.

Given a CGF (E, P), we define a corresponding P/T net $N = (S, T)$ and a corresponding marked P/T net $N(m_0)$. We first need to introduce an auxiliary function $Mark(P)$ that associates to a solution P the multiset of its variables:

$$Mark(P) \;=\; \begin{cases} \emptyset & \text{if } P = \mathbf{0} \\ \{X\} \oplus Mark(P') & \text{if } P = X|P' \end{cases}$$

Definition 6 (Net of a CGF). *Given a CGF* (E, P), *with* $Net_{(E,P)}$ *we denote the corresponding P/T net* (S, T) *where:*

$$\begin{aligned} S &= \{X \mid X \text{ occurs in } E\} \\ T &= \big\{\ \{X\} \Rightarrow Mark(X_1|\cdots|X_n) \mid \\ &\qquad E.X.i = \tau_{(r)}; (X_1|\cdots|X_n)\ \big\} \cup \\ &\quad \big\{\ \{X, Y\} \Rightarrow Mark(X_1|\cdots|X_n) \oplus Mark(Y_1|\cdots|Y_m) \mid \\ &\qquad E.X.i = ?a_{(r)}; (X_1|\cdots|X_n) \text{ and } E.Y.j = !a_{(r)}; (Y_1|\cdots|Y_m)\ \big\} \end{aligned}$$

The corresponding marked P/T net is $Net_{(E,P)}(Mark(P))$.

Note that the set of places S corresponds to the set of variables X defined in E, the transitions represents the possible actions, and the initial marking is the multiset of variables in the solution P. It is also worth observing that in the net semantics we do not consider the rates (r) of the actions.

We now formalize the correspondence between the behaviors of a CGF and of its corresponding P/T net.

Theorem 1. *Consider a CGF* (E, P) *and the corresponding P/T net* $Net_{(E,P)} = (S, T)$. *We have that:*

1. *if* $\langle l : P^\dagger \xrightarrow{r} Q^\dagger \rangle$ *is in* $Next(E, P)$ *(for some* l *and* r*) then we have also that* $Mark(P)[\rangle Mark(Q)$ *in* $Net_{(E,P)}$;
2. *if there exists* m *such that* $Mark(P)[\rangle m$ *in* $Net_{(E,P)}$, *then there exist* l, r *and* Q *such that* $\langle l : P^\dagger \xrightarrow{r} Q^\dagger \rangle$ *is in* $Next(E, P)$ *and* $Mark(Q) = m$.

Proof (sketch). The proofs of the two statements are by case analysis on the possible transitions in $Next(E, P)$ as defined in the Definition 2 —for the first statement— or on the possible transitions enabled in $Mark(P)$ as defined in the Definition 6 —for the second statement. □

This theorem allows us to conclude that the P/T net semantics faithfully reproduces the standard CGF transitions. The only difference is that it abstracts away from the stochastic rates. For this reason, we consider the P/T net semantics as a purely nondeterministic interpretation of CGF. Reachability, boundedness, termination, and divergence are decidable for P/T nets; thus we can conclude that all these properties are decidable also in the CGF under a purely nondeterministic interpretation.

As a consequence of Theorem 1, existential termination is decidable.

Theorem 2. *Consider a CGF* (E, P)*. We have that* (E, P) *existentially terminates if and only if a dead marking is reachable in the* $Net_{(E,P)}(Mark(P))$*.*

Proof. It is easy to see from the definition of $LTG(E, P)$ and $|LTG(E, P)|$ that the latter contains all and only those solutions (in normal form) reachable in (E, P) with a finite number of transitions, each one having a probability > 0 to be chosen. Thus a solution is reachable with probability > 0 if and only if it is in $|LTG(E, P)|$. As a consequence of Theorem 1 we have that a solution $Q^{\dagger}$ is in $|LTG(E, P)|$ if and only if $Mark(Q)$ is reachable in $Net_{(E,P)}(Mark(P))$. Moreover, Theorem 1 also guarantees that Q is terminated if and only if $Mark(Q)$ is a dead marking in $Net_{(E,P)}$ (this proves the theorem). □

As a corollary of Theorem 1 we have that also the probabilistic variants of reachability and boundedness can be reduced to the corresponding properties in the nondeterministic setting. On the contrary, this does not hold for divergence. (This will be discussed in the next section.) We can summarize the results of this section simply saying that ϵ-*termination*, ϵ-*reachability*, and ϵ-*boundedness* are decidable when $\epsilon = 0$.

4 Undecidability Results

This section is divided in two parts. In the first one we prove that probabilistic termination (i.e. ϵ-termination with $\epsilon > 0$) is undecidable. (We also comment on how to show that also ϵ-divergence, ϵ-boundedness, and ϵ-reachability are undecidable when $\epsilon > 0$.) In the second part we prove the undecidability of universal termination (thus also of 0-divergence).

4.1 Undecidability of Probabilistic Termination

We prove the undecidability of probabilistic termination showing how to approximately model in CGF the behavior of any Random Access Machines (RAMs) [7], a well known register based Turing powerful formalism. More precisely, we reduce

the termination problem for RAMs to the probabilistic termination with probability higher than any ϵ such that $0 < \epsilon < 1$.

We first recall the definition of Random Access Machines.

Definition 7 (Random Access Machines (RAMs)). *A RAM $\mathcal{R}$ is composed of a set of registers $r_1, \cdots, r_m$ that contain non negative integer numbers and a set of indexed instructions $I_1, \cdots, I_n$ of two possible kinds:*

- *$I_i = Inc(r_j)$ that increments the register r_j and then moves to the execution of the instruction with index $i + 1$ and*
- *$I_i = DecJump(r_j, s)$ that attempts to decrement the register r_j; if the register does not hold 0 then the register is actually decremented and the next instruction is the one with index $i + 1$, otherwise the next instruction is the one with index s.*

We use the following notation: $(I_i, r_1 = l_1, \cdots, r_m = l_m)$ represents the state of the computation of the RAM which is going to execute the instruction I_i with registers that contain $l_1, \cdots, l_m$, respectively; $(I_i, r_1 = l_1, \cdots, r_m = l_m) \mapsto (I_j, r_1 = l'_1, \cdots, r_m = l'_m)$ describes one step of computation of the RAM; $(I_i, r_1 = l_1, \cdots, r_m = l_m) \downarrow$ denotes final states of the computation in which I_i is undefined. Without loss of generality, we assume the existence of a special index *halt* such that all final states contain an instruction with that index, namely $(I_i, r_1 = l_1, \cdots, r_m = l_m) \downarrow$ if and only if $i = halt$.

The basic idea that we follow in modeling RAMs in CGF is to use one species I^i for each instruction I_i, and one species R^j for each register r_j. The state $(I_i, r_1 = l_1, \cdots, r_m = l_m)$ of the RAM is modeled by a solution that contains one molecule of species I^i, l_1 molecules of species R^1, $\cdots$, and l_m molecules of species R^m (plus a certain amount of inhibitor molecules of species Inh, whose function will be discussed below). The behavior of the molecules of species I^i is to update the register according to the corresponding instruction I_i, and to activate the execution of the next instruction I_j by producing the molecule of species I^j.

An $Inc(r_j)$ instruction simply produces one molecule of species R^j. On the other hand, a $DecJump(r_j, s)$ instruction should test the absence of molecules of species R^j before deciding whether to execute the jump, or to consume one of the available molecules of that species. As it is not possible to verify the absence of molecules, we admit the execution of the jump even if molecules of species R^j are available. In this case, we say that a *wrong jump* is executed. In order to reduce the probability of wrong jumps, we put their execution in competition with alternative behaviors involving the inhibitor molecules in such a way that the greater is the quantity of inhibitor molecules in the solution, the smaller is the probability to execute a wrong jump.

We are now ready to formally define our encoding of RAMs.

Definition 8. *Given a RAM $\mathcal{R}$ and one of its states $(I_i, r_1 = l_1, \cdots, r_m = l_m)$, let $[\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h$ denote the solution:*

$$I^i \mid \prod_{l_1} R^1 \mid \cdots \mid \prod_{l_m} R^m \mid \prod_{h} Inh$$

where:

$$I^i = \begin{cases} \tau;(I^{i+1}|R_j) & \text{if } I_i = Inc(r_j) \\ !r_j;(I^{i+1}|Inh) \oplus \tau;C^2_{i,s} & \text{if } I_i = DecJump(r_j,s) \\ \mathbf{0} & \text{if } I_i = I_{halt} \end{cases}$$

$$C^2_{i,s} = !inh;I^i \oplus \tau;C^1_{i,s} \qquad C^1_{i,s} = !inh;I^i \oplus \tau;I^s$$

$$R^j = ?r_j;\mathbf{0} \qquad Inh = ?inh;Inh$$

Note that h is used to denote the number of occurrences of the molecules of species Inh. We take all subscripts action rates equal to 1 and we omit them (this choice allows us to simplify the proof of Proposition 1). In the following, we use $E_{\mathcal{R}}$ for the set of the above definitions of species I^i, $C^2_{i,s}$, $C^1_{i,s}$, R^j, Inh.

Note that before actually executing a jump, two internal τ actions must be executed in sequence (those in the definition of the species $C^2_{i,s}$ and $C^1_{i,s}$), and both of them are in competition with the action $!inh$ willing to perform an interaction with one of the inhibitor molecules of species Inh. Thus, the higher is the number of inhibitor molecules, the smaller is the probability to perform this sequence of two internal actions.

We now formalize the correspondence between the behavior of a RAM and of its encoding in CGF.

Proposition 1. *Let $\mathcal{R}$ be a RAM. Given one of its states $(I_i, r_1 = l_1, \cdots, r_m = l_m)$ and $[\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h$, for any h, we have:*

1. *if $I_i = I_{halt}$ then $(I_i, r_1 = l_1, \cdots, r_m = l_m) \downarrow$ and $Next(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)$ has no transitions;*
2. *if $I_i = Incr_j$ or $I_i = DecJump(r_j, s)$ with $l_j = 0$ and $(I_i, r_1 = l_1, \cdots, r_m = l_m) \mapsto (I_j, r_1 = l'_1, \cdots, r_m = l'_m)$, then the solution $[\![(I_j, r_1 = l'_1, \cdots, r_m = l'_m)]\!]^\dagger_h$ is reachable in $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)$ with probability $= 1$;*
3. *if $I_i = DecJump(r_j, s)$ with $l_j > 0$ and $(I_i, r_1 = l_1, \cdots, r_m = l_m) \mapsto (I_j, r_1 = l'_1, \cdots, r_m = l'_m)$, then the solution $[\![(I_j, r_1 = l'_1, \cdots, r_m = l'_m)]\!]^\dagger_{h+1}$ is reachable in $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)$ with probability $> 1 - \frac{1}{h^2}$.*

Proof. If $I_i = I_{halt}$ or $I_i = Inc(r_j)$ the corresponding statements (the first one and the first part of the second one) are easy to prove. We detail the proof only for $I_i = DecJump(r_j, s)$.

If r_j is empty, the probability measure for the computations in $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)$ passing through $[\![(I_j, r_1 = l'_1, \cdots, r_m = l'_m)]\!]^\dagger_h$ is (see Figure 1):

$$\sum_{i=0}^{\infty} \left(\frac{h}{h+1} + \frac{1}{h+1} \times \frac{h}{h+1}\right)^i \times \frac{1}{(h+1)^2} = 1$$

If r_j is not empty, i.e. $l_j > 0$, the standard probability measure for the computations passing through $[\![(I_j, r_1 = l'_1, \cdots, r_m = l'_m)]\!]^\dagger_{h+1}$ is (see Figure 1):

$$\sum_{i=0}^{\infty} \left(\frac{1}{l_j+1} \times \frac{h}{h+1} + \frac{1}{l_j+1} \times \frac{1}{h+1} \times \frac{h}{h+1}\right)^i \times \frac{l_j}{l_j+1} > 1 - \frac{1}{h^2} \qquad \square$$

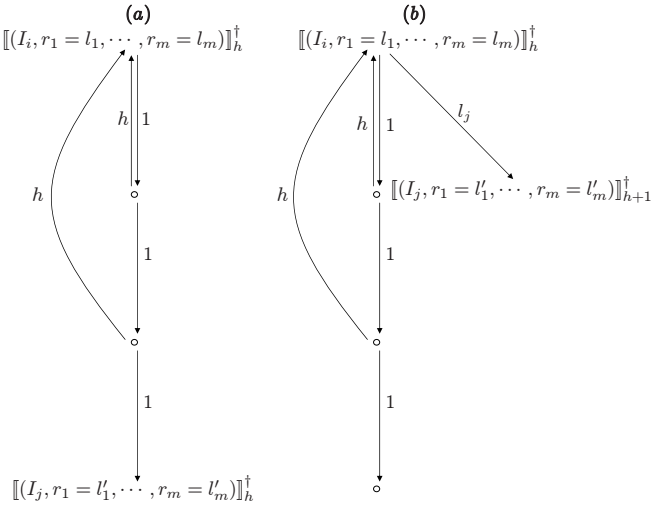


Fig. 1. Fragment of the CTMC $|LTG(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)|$ in case $I_i = DecJump(r_j, s)$ with $l_j = 0$ (a) or $l_j > 0$ (b)

The above proposition states the correspondence between a single RAM step and the corresponding encoding in CGF. We conclude that a RAM terminates its computation if and only if a terminated solution is reachable with a probability that depends on the initial number of inhibitor molecules in the encoding.

Theorem 3. *Let $\mathcal{R}$ be a RAM. We have that the computation of $\mathcal{R}$ starting from the state $(I_i, r_1 = l_1, \cdots, r_m = l_m)$ terminates if and only if the CGF $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)$ probabilistically terminates with probability higher than $1 - \sum_{k=h}^{\infty} \frac{1}{k^2}$.*

Proof. In the light of Proposition 1 we have that only decrement operations are not reproduced with probability $= 1$, but with probability $> 1 - \frac{1}{h^2}$. Moreover, after the execution of a decrement operation, the value h of inhibitor molecules is incremented by one. Thus, a RAM computation including d decrement operations is faithfully reproduced with probability strictly greater than $\prod_{k=h}^{h+d} \left(1 - \frac{1}{k^2}\right) > 1 - \sum_{k=h}^{h+d} \frac{1}{k^2}$. Henceforth, any terminating computation is reproduced with probability strictly greater than $1 - \sum_{k=h}^{\infty} \frac{1}{k^2}$. □

It is well known that the series $\sum_{h=1}^{\infty} \frac{1}{h^2}$ is convergent (to $\frac{\pi^2}{6}$), thus for every small value $\delta > 0$ there exists a corresponding initial amount h of inhibitor molecules such that $\sum_{k=h}^{\infty} \frac{1}{k^2} < \delta$. Henceforth, in order to reduce RAM termination to probabilistic termination with probability higher than any $0 < \epsilon < 1$, it is sufficient to consider an initial value h such that $\sum_{k=h}^{\infty} \frac{1}{k^2} < (1 - \epsilon)$.

The RAM encoding presented in Definition 8 reproduces also unbounded RAM computations with any degree of precision. Thus also ϵ-divergence is undecidable when $\epsilon > 0$. On the contrary, such encoding does not allow us to prove the undecidability of ϵ-boundedness and ϵ-reachability.

We first show how to reduce the RAM divergence problem to ϵ-boundedness. This does not hold for the encoding in the Definition 8 because there exists divergent RAMs with a bounded corresponding CGF. Consider, for instance, the RAM composed of only the instruction $I_1 = DecJump(r_1, 1)$ that performs an infinite loop if the register r_1 is initially empty. It is easy to see that the corresponding CGF is bounded.

In order to guarantee that an infinite RAM computation generates an unbounded CGF, we can simply add a new molecule of a new species A every time a jump is performed. As an infinite RAM computation executes infinitely many jump operations, an unbounded amount of molecules of species A will be generated. The new encoding is defined as in the Definition 8 replacing the definition of the species $C^1_{i,s}$ with the following one: $C^1_{i,s} = !inh; I^i \oplus \tau; (I^s|A)$.

We conclude this section observing how to reduce RAM termination to ϵ-reachability. This does not hold for the above encodings because the solution representing the final state of a RAM computation is not known beforehand. In fact, besides the fact that the final contents of the registers is not known, we have that the final solution will contain a number of inhibitor molecules that depends on the number of decrement operations executed during the computation (as each decrement adds one molecule of species Inh). In order to know beforehand the final solution, we allow the molecule I^{halt} to remove the register molecules of species R^j as well as all the inhibitor molecules of species Inh. In this way, if the computation terminates, we have that the final solution surely contains only the molecule I^{halt}.

Namely, we modify in the Definition 8 the definitions of the species I^{halt} and Inh as follows:

$$\begin{aligned} I^{halt} &= \bigoplus_{j=1}^{m} !r_j; I^{halt} \oplus !remove; I^{halt} \\ Inh &= ?inh; Inh \oplus ?remove; \mathbf{0}. \end{aligned}$$

4.2 Undecidability of Universal Termination

The undecidability of universal termination (thus also of 0-divergence) is proved introducing an intermediary nondeterministic computational model, that we call *finitely faulting RAMs* ($\mathcal{FF}$RAMs). This model corresponds to RAMs in which the execution of *DecJump* instructions is nondeterministic when the tested register is not empty: an $\mathcal{FF}$RAM can either decrement the register or execute a wrong jump. The peculiarity of $\mathcal{FF}$RAMs is that in an infinite computation only finitely many wrong jumps are executed. We first show that it is possible to define an encoding of $\mathcal{FF}$RAMs in CGF such that the universal termination problem for $\mathcal{FF}$RAMs coincides with the universal termination problem for the corresponding CGF. Then we prove the undecidability of the universal termination problem for $\mathcal{FF}$RAMs showing how to reduce the RAM termination problem to the verification of the existence of an infinite computation in

$\mathcal{FF}$RAMs (which corresponds to the complement of the universal termination problem). We start defining *finitely faulting RAMs*.

Definition 9 (Finitely Faulting RAMs ($\mathcal{FF}$RAMs)). Finitely Faulting RAMs *are defined as traditional RAMs (see Definition 7) with the only difference that given an instruction* $I_i = DecJump(r_j, s)$ *and a RAM state* $(I_i, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)$ *with* $l_j > 0$*, two possible computation steps are permitted:* $(I_i, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m) \mapsto (I_{i+1}, r_1 = l_1, \cdots, r_j = l_j - 1, \cdots, r_m = l_m)$ *and* $(I_i, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m) \mapsto (I_s, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)$*. The second computation step is called* wrong jump *because a jump is executed even if the tested register is not empty. The peculiar property of $\mathcal{FF}$RAMs is that in every computation (also infinite ones), finitely many wrong jumps are executed.*

We now show how to define an encoding of $\mathcal{FF}$RAMs in CGF such that infinite computations in the $\mathcal{FF}$RAMs computational model corresponds to infinite computation with probability > 0 in the corresponding CGF.

The $\mathcal{FF}$RAM encoding is defined as in Definition 8 adding a transition to a terminated state which can be selected with probability $\geq \frac{1}{2}$ while executing wrong jumps. In this way, we guarantee that in an infinite computation infinitely many wrong jumps cannot be executed because the new transition to the terminated state cannot be avoided indefinitely.

Definition 10 ($\mathcal{FF}$RAM Modeling). *Given a $\mathcal{FF}$RAM $\mathcal{R}$ and one of its states* $(I_i, r_1 = l_1, \cdots, r_m = l_m)$*,* $[\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h$ *is defined as in Definition 8. Also the species* I^i*,* R^j*,* Inh*, and* $C^2_{i,s}$ *are defined as in Definition 8, while* $C^1_{i,s}$ *is defined as follows:*

$$C^1_{i,s} = !inh; I^i \oplus \tau; C^0_s \qquad C^0_s = !r_j; I_{halt} \oplus \tau; I_s$$

In the following, we use $E_{\mathcal{R}}$ *for the new set of definitions of species* I^i*,* $C^2_{i,s}$*,* $C^1_{i,s}$*,* C^0_s*,* R^j*, and* Inh*.*

We now revisit the Proposition 1 adapting it to the new encoding.

Proposition 2. *Let $\mathcal{R}$ be a $\mathcal{FF}$RAM. Given one of its states* $(I_i, r_1 = l_1, \cdots, r_m = l_m)$ *and* $[\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h$*, for any h, we have:*

1. *(as in Proposition 1);*
2. *(as in Proposition 1);*
3. *if* $I_i = DecJump(r_j, s)$ *with* $l_j > 0$ *then with probability* 1 *one of the following states are reachable in* $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)]\!]_h)$*:*
   - $[\![(I_{i+1}, r_1 = l_1, \cdots, r_j = l_j - 1, \cdots, r_m = l_m)]\!]^{\dagger}_{h+1}$ *(with prob.* $> 1 - \frac{1}{h^2}$*);*
   - $[\![(I_{halt}, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)]\!]^{\dagger}_{h}$*;*
   - $[\![(I_s, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)]\!]^{\dagger}_{h}$ *(with probability* $0 < p < \frac{1}{2}$*).*

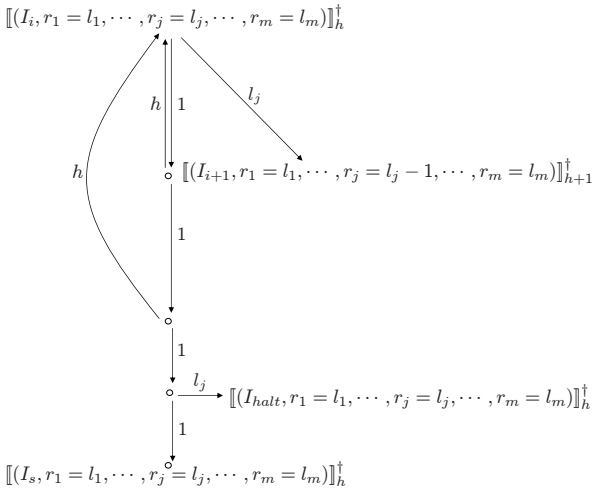


Fig. 2. Fragment of the CTMC $|LTG(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)|$ in case $I_i = DecJump(r_j, s)$ with $l_j > 0$

Proof. The first two statements are proved as in Proposition 1. We sketch the proof for the third statement. The probability measure for the computations in $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)]\!]_h)$ passing through $[\![(I_{i+1}, r_1 = l_1, \cdots, r_j = l_j - 1, \cdots, r_m = l_m)]\!]^\dagger_{h+1}$ is computed as in Proposition 1.

The probability measure p for the computations passing through $[\![(I_s, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)]\!]^\dagger_h$ is (see Figure 2):

$$\sum_{i=0}^{\infty} \left(\frac{1}{l_j+1} \times \frac{h}{h+1} + \frac{1}{l_j+1} \times \frac{1}{h+1} \times \frac{h}{h+1}\right)^i \times \left(\frac{1}{l_j+1}\right)^2 \times \left(\frac{1}{h+1}\right)^2$$

It is easy to see that as $l_j > 0$, then $p < \frac{1}{2}$. Finally, we observe that the probability measure of the computations leading to $[\![(I_{halt}, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)]\!]^\dagger_h$ is equal to 1 minus the probability measure of the computations passing through either $[\![(I_{i+1}, r_1 = l_1, \cdots, r_j = l_j - 1, \cdots, r_m = l_m)]\!]^\dagger_{h+1}$ or $[\![(I_s, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)]\!]^\dagger_h$. □

The above proposition states the correspondence between a single computation step of a $\mathcal{FF}$RAM and that of the corresponding CGF. We conclude that a $\mathcal{FF}$RAM has an infinite computation if and only if there exists an infinite computation with probability > 0 in the corresponding CGF.

Theorem 4. *Let $\mathcal{R}$ be a $\mathcal{FF}$RAM. We have that $\mathcal{R}$ has an infinite computation starting from the state $(I_i, r_1 = l_1, \cdots, r_m = l_m)$ if and only if the CGF $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)$ has an infinite computation for some initial amount h of inhibitor molecules.*

Proof. We first consider the *only if* part. Assume the existence of an infinite computation of $\mathcal{R}$ starting from the state $(I_i, r_1 = l_1, \cdots, r_m = l_m)$. This computation will execute infinitely many $DecJump$ instructions, but only finitely many wrong jumps. We now consider the CGF $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)$ for a generic h. According to the Proposition 2, it can reproduce the same infinite computation with probability strictly greater than $\prod_{k=h}^{\infty}\left(1-\frac{1}{k^2}\right) \times \prod_{k=1}^{w} p_k$ where w is the number of wrong jumps, and p_k is the probability for the k-th wrong jump computed as in Proposition 2. Let p_{min} be the minimum among $p_1, \cdots, p_w$. We have that the above probability is strictly greater than $\left(1-\sum_{k=h}^{\infty}\frac{1}{k^2}\right) \times \left(\frac{1}{p_{min}}\right)^w$. We have already discussed, after Theorem 3, that the series $\sum_{h=1}^{\infty}\frac{1}{h^2}$ is convergent, thus there exists h such that $\sum_{k=h}^{\infty}\frac{1}{k^2} < 1$. If we consider this particular value h, the overall probability for the infinite computation is > 0.

We now consider the *if* part. Assume the existence of an infinite computation with probability > 0 in the CGF $(E_{\mathcal{R}}, [\![(I_i, r_1 = l_1, \cdots, r_m = l_m)]\!]_h)$ for some h. This computation corresponds to an infinite computation of $\mathcal{R}$ for the two following reasons. We first observe that the infinite computation reproduces infinitely many correct computation steps $(I_i, r_1 = l_1, \cdots, r_m = l_m) \mapsto (I_j, r_1 = l'_1, \cdots, r_m = l'_m)$ of $\mathcal{R}$. In fact, the unique wrong computation step could be the one described in the second item of the third statement of Proposition 2. This computation step leads to the encoding of the terminated state $(I_{halt}, r_1 = l_1, \cdots, r_j = l_j, \cdots, r_m = l_m)$, but in this case the computation cannot be infinite. Then, we observe that the number of wrong jumps is finite. In fact, if we assume (by contradiction) that the computation contains infinitely many wrong jumps, we have that (in the light of the third item of the third statement of Proposition 2) the probability of the infinite computation is smaller than $\prod_{i=1}^{\infty}\frac{1}{2}$, thus it cannot be > 0. □

We now prove that the existence of an infinite computation in $\mathcal{FF}$RAMs is undecidable by defining an encoding that reduces the termination problem for RAMs to the divergence problem for $\mathcal{FF}$RAMs. As it is not restrictive, we consider only RAMs starting with all registers empty. Our technique has been inspired by a similar one used in [3]. We initially assume that an arbitrary number k of wrong jumps occurs and, as a consequence, the number k is introduced in a special register. Then we let the $\mathcal{FF}$RAM repeat indefinitely the simulation of the behavior of the corresponding RAM, but if this simulation requires more than k steps, the encoding blocks (this is ensured by decrementing the special register before simulating every computational step). In this way, if a RAM terminates, then the corresponding $\mathcal{FF}$RAM (with k greater than the length of the RAM computation) can diverge. On the other hand, if an infinite computation of the $\mathcal{FF}$RAM exists, this has an infinite suffix that does not contain wrong jumps. In this correct part of the computation, the encoding faithfully simulates the RAM computation infinitely often; this is possible only if the RAM terminates.

Theorem 5. *Given a RAM $\mathcal{R}$, there exists a corresponding $\mathcal{FF}$RAM $[\![\mathcal{R}]\!]$ such that the computation of $\mathcal{R}$ (starting with all registers empty) terminates if and only if $[\![\mathcal{R}]\!]$ has an infinite computation (starting with all registers empty).*

Proof. Given a RAM $\mathcal{R}$ with instructions $I_1, \cdots, I_n$ (assuming $I_n = I_{halt}$) and registers $r_0, \cdots, r_m$, with $[\![\mathcal{R}]\!]$ we denote the $\mathcal{FF}$RAM composed of the registers $r_0, r_1, \cdots, r_m, r_{m+1}, r_{m+2}, r_{m+3}$ and of the following instructions:

$$
\begin{array}{lcll}
J_1 & = & Inc(r_{m+1}) & \\
J_2 & = & DecJump(r_{m+1}, 1) & \\
J_{3i} & = & DecJump(r_{m+1}, halt) & (\text{for } 1 \leq i < n) \\
J_{3i+1} & = & Inc(r_{m+2}) & (\text{for } 1 \leq i < n) \\
J_{3i+2} & = & \begin{cases} Inc(r_j) & \text{if } I_i = Inc(r_j) \\ DecJump(r_j, 3s) & \text{if } I_i = DecJump(r_j, s) \end{cases} & (\text{for } 1 \leq i < n) \\
J_{3n+2j} & = & DecJump(r_j, 3n + 2j + 2) & (\text{for } 1 \leq i \leq m) \\
J_{3n+2j+1} & = & DecJump(r_{m+3}, 3n + 2j) & (\text{for } 1 \leq i \leq m) \\
J_{3n+2m+2} & = & DecJump(r_{m+2}, 3) & \\
J_{3n+2m+3} & = & Inc(r_{m+1}) & \\
J_{3n+2m+4} & = & DecJump(r_{m+3}, 3n + 2m + 2) &
\end{array}
$$

We prove that the computation of $\mathcal{R}$ starting from the state $(I_1, r_0 = 0, \cdots, r_m = 0)$ terminates if and only if $[\![\mathcal{R}]\!]$ has an infinite computation starting from the state $(J_1, r_0 = 0, \cdots, r_{m+3} = 0)$.

We first consider the *only-if* part. We assume that the RAM $\mathcal{R}$ terminates after the execution of k steps. The corresponding $\mathcal{FF}$RAM $[\![\mathcal{R}]\!]$ has the following infinite computation which contains exactly k wrong jumps. The wrong jumps are all executed at the beginning of the computation in order to introduce in r_{m+1} the value k. Then the computation proceeds simulating infinitely many times the computation of $\mathcal{R}$. Note that at the end of each simulation, all the registers $r_1, \cdots, r_m$ are emptied, and the value k (which is introduced in r_{m+2} during the computation, is moved back in r_{m+1}). Note also that the register r_{m+3} is always empty, and that it is simply tested for zero by instructions that must always perform a jump.

We now consider the *if* part. Assume that the $\mathcal{FF}$RAM $[\![\mathcal{R}]\!]$ has an infinite computation. This computation starts with k executions of the instructions J_1 and J_2. The loop between these two instructions cannot proceed indefinitely as it contains a wrong jump. At the end of this first phase, the register r_{m+1} contains k. Then the computation continues by simulating the behavior of the RAM $\mathcal{R}$, and before executing every instruction the register r_{m+1} is decremented and the register r_{m+2} is incremented. If (by contradiction) the register r_{m+1} becomes empty before completing the simulation of $\mathcal{R}$, the computation should block. Thus, the simulation completes before simulating k steps. After, all the registers $r_0, \cdots, r_m$ are emptied, the value k is reintroduced in r_{m+1}, and a new simulation is started. This part of the computation, i.e. simulation of $\mathcal{R}$ and subsequent reset of the registers, surely terminates because the simulation of $\mathcal{R}$ includes at most k steps, and the subsequent reset of the registers cannot proceed indefinitely. We can conclude that an infinite computation includes infinitely many simulations of the computation of $\mathcal{R}$ and, as an $\mathcal{FF}$RAM can perform only finitely many wrong jumps, infinitely many of these simulations are correct. This implies that the RAM $\mathcal{R}$ terminates within k computation steps. □

5 Conclusion

In this paper we have investigated the decidability of termination problems in CGF, a process algebra proposed in [2] for the compositional description of chemical systems. In particular, we have proved that existential termination is decidable, probabilistic termination is undecidable, and universal termination is decidable under a purely nondeterministic interpretation of CGF while it turns to be undecidable under the stochastic semantics.

It is worth saying that similar results hold also for lossy channels: universal termination is decidable in lossy channels while it turns out to be undecidable in their *probabilistic* variant [1]. Nevertheless, the result on lossy channels is not comparable with ours. In fact, in CGF process communication is synchronous (in lossy channels synchronous communication is not admitted) while in the lossy channel model it is asynchronous through unbounded FIFO buffers (that cannot be directly encoded in CGF).

Acknowledgement. We would like to acknowledge M. Bravetti, D. Soloveichik, H. Wiklicky, E. Winfree, and the anonymous referees for their insightful comments on previous versions of this paper.

References

1. Abdulla, P., Baier, C., Iyer, P., Jonsson, B.: Reasoning about Probabilistic Lossy Channel Systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 320–333. Springer, Heidelberg (2000)
2. Cardelli, L.: On Process Rate Semantics. Theoretical Computer Science (in press, 2008), http://dx.doi.org/10.1016/j.tcs.2007.11.012
3. Carstensen, H.: Decidability Questions for Fairness in Petri Nets. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 396–407. Springer, Heidelberg (1987)
4. Esparza, J., Nielsen, M.: Decidability Issues for Petri Nets, Technical report BRICS RS-94-8 (1994)
5. Kemeny, J.G., Snell, J.L., Knapp, A.W.: Denumerable Markov Chains. Springer, Heidelberg (1976)
6. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
7. Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Englewood Cliffs (1967)
8. Soloveichik, D., Cook, M., Winfree, E., Bruck, J.: Computation with Finite Stochastic Chemical Reaction Networks. Natural Computing (in press, 2008), http://dx.doi.org/10.1007/s11047-008-9067-y

Towards a Unified Approach to Encodability and Separation Results for Process Calculi

Daniele Gorla

Dip. di Informatica, Univ. di Roma "La Sapienza"
PPS - Université Paris Diderot & CNRS, France

Abstract. In this paper, we present a unified approach to evaluating the relative expressive power of process calculi. In particular, we identify a small set of criteria (that have already been somehow presented in the literature) that an encoding should satisfy to be considered a good means for language comparison. We argue that the combination of such criteria is a valid proposal by noting that: (*i*) the best known encodings appeared in the literature satisfy them; (*ii*) this notion is not trivial, because there exist encodings that do not satisfy all the criteria we have proposed; (*iii*) the best known separation results can be formulated in terms of our criteria; and (*iv*) some widely believed (but never formally proved) separation results can be proved by using the criteria we propose. Moreover, the way in which we prove known separation results is easier and more uniform than the way in which such results were originally proved.

1 Introduction

As argued in [27], one of the hottest topic in concurrency theory, and mainly in process calculi, is the identification of a uniform way to formally compare different languages from the expressiveness point of view. Indeed, while the literature contains several results and claims concerning the expressive power of a language, such results are usually difficult to appreciate because they are proved sound by using different criteria. For a very good overview of the problem, we refer the reader to [31].

In the 1980s, the trend was to adopt the approach followed in computability theory and study the *absolute* expressive power of languages, e.g. by studying which problems were solvable or which operators were definable in a given language. In the 1990s, the focus moved to the *relative* expressive power: it became more interesting to understand the extent to which a language could be encoded in another one, also because of the proliferation of different process calculi.

A very common approach to proving soundness of encodings is based on the notion of *full abstraction*. This concept was introduced in the 1970s to require an exact correspondence between a denotational semantics of a program and its operational semantics. Intuitively, a denotational semantics is fully abstract if it holds that two observably equivalent programs (i.e., two programs that 'behave in the same way' in any execution context) have the same denotation, and vice versa. The notion of full abstraction has been adapted to prove soundness of encodings by requiring that an encoding maps equivalent source terms into equivalent target terms, and vice versa. This adaptation was justified by the fact that an encoding resembles a denotation function: they both map

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 492–507, 2008.

elements of a formalism (viz., terms of the source language) into elements of a different formalism (another language, in the case of an encoding, or a mathematical object, in the case of a denotation function). In this way, the stress is put on the requirement that the encoding must translate a language in another one while respecting some associated equivalences. This can be very attractive, e.g., if in the target we can exploit automatic tools to prove equivalences and then pull back the obtained result to the source. However, we believe that full abstraction is too focused on the equivalences and thus it gives very little information on the computation capabilities of the two languages.

Operational and structural criteria have been developed in the years to state and prove separation results [9,17,29,32,33], that are a crucial aspect of building a hierarchy of languages. Indeed, to prove that a language $\mathcal{L}_1$ is more expressive than another language $\mathcal{L}_2$, we need to show that there exists a "good" encoding of the latter in the former, but not vice versa. Usually, the latter fact is very difficult to prove and is obtained by: (1) identifying a problem that can be solved in $\mathcal{L}_1$ but not in $\mathcal{L}_2$, and (2) finding the least set of criteria that an encoding should meet to translate a solution in $\mathcal{L}_1$ into a solution in $\mathcal{L}_2$. Such criteria are problem-driven, in that different problems call for different criteria (compare, for example, the criteria in [29,32,33] with those in [9,17]). Moreover, the criteria used to prove separation results are usually not enough to testify to the quality of an encoding: they are considered minimal requirements that any encoding should satisfy to be considered a good means for language comparison.

In this paper, we present a new proposal for assessing the quality of an encoding, tailored to aspects that are strictly related to relative expressiveness. We isolate a small set of requirements that, in our opinion, are very well-suited to proving both soundness of encodings and separation results. In this way, we obtain a notion of encodability that can be used to place two (or more) languages in a clearly organized hierarchy. A preliminary proposal appeared in [15] but it was formulated in a too demanding way.

Of course, in order to support our proposal, we have to give evidences of its reasonableness. To this aim, we exhibit both philosophical and pragmatic arguments. From the pragmatic side, we notice that most of the best known encodings appeared in the literature satisfy our criteria and that their combination is not trivial, because there exist some encodings (namely, the encodings of π-calculus in Mobile Ambients proposed in [10,11]) that do not satisfy all the criteria we propose. Moreover, we also prove that the best known separation results can be formulated and proved (in an easier and more uniform way) in terms of our criteria; furthermost, some widely believed (but never proved) separation results can be now formally proved by using the criteria we propose (this task is carried out in [14]). The philosophical part is, instead, more delicate because we have to convince the reader that every proposed criterion is deeply related to relative expressiveness. To this aim, we split the criteria in two groups: structural and semantic. We think that structural criteria are difficult to criticize: we simply require that the encoding is compositional and that it does not depend on the specific names appearing in the source term. Semantic criteria are, as usual, more debatable, because different people have different views on the semantics of a calculus and because the same semantic notions can be defined in different ways. Here, we assume that an encoding should be: *operationally corresponding*, in the sense that it preserves and reflects the computations of the source terms; *divergence reflecting*, in that we do not want to turn a terminating

term into a non-terminating one; and *success sensitive*, i.e., once defined a notion of successful computation of a term, we require that successful source terms are mapped into successful target terms and vice versa.

Although intuitively quite clear, the above mentioned criteria can be formulated in different ways. In particular, operational correspondence is usually defined up to some semantic equivalence/preorder that gets rid of dead processes yielded by the encoding. However, there is a wide range of equivalences/preorders and choosing one or another is always highly debatable. In Section 2 we start by leaving the notion of equivalence/preorder unspecified; this is, in our opinion, the ideal scenario, where encodability and separation results do not depend on the particular semantic theory chosen. However, when we want to prove some concrete result, we are forced to make assumptions on the equivalence used in operational correspondence. In doing this, we try to work at the highest possible abstraction level; in particular, we never commit to any specific equivalence/preorder and always consider meaningful families of such relations.

The paper is organized as follows. In Section 2, we present the criteria that we are going to consider and compare them with other ones already presented in the literature. Then, in Section 3, we show how to prove (in a simpler and more uniform way) known separation results appearing in the literature; to this aim, we specialize in three ways the semantic theory used to define operational correspondence. In Section 4, we conclude by summing up our main contributions and discussing future work.

For space limitations, we shall work with process calculi (CCS [22]; the asynchronous π-calculus, π_a [5]; the separate and mixed choice π-calculus, π_{sep} and π_{mix} [35]; Mobile Ambients, MA [11]; the π-calculus with polyadic synchronizations, e^π and π^n [9]) without defining them; their syntax and operational semantics can be found in the on-line version of this paper or in the cited references.

2 The Encodability Criteria

In this section we discuss the criteria an encoding should satisfy to be considered a good means for language comparison. For the moment, we work at an abstract level and do not commit to any precise formalism. Indeed, we just assume a (countable) set of names $\mathcal{N}$ and specify a calculus as a triple $\mathcal{L} = (\mathcal{P}, \longmapsto, \asymp)$, where

- $\mathcal{P}$ is the set of language terms (usually called *processes*) that is built up from the terminated process $\mathbf{0}$ by at least using the parallel composition operator '|'.
- $\longmapsto$ is the operational semantics, needed to specify how a process computes; following common trends in process calculi, we specify the operational semantics by means of *reductions*. As usual, $\Longmapsto$ denotes the reflexive and transitive closure of $\longmapsto$. To compositionally reason on process reductions, we shall also assume a labeled transition relation, $\stackrel{\mu}{\rightarrow}$, whose τ's characterize $\longmapsto$.
- $\asymp$ is a behavioural equivalence/preorder, needed to describe the abstract behaviour of a process. Usually, $\asymp$ is a congruence at least with respect to parallel composition; it is often defined in the form of a barbed equivalence [25] or can be derived directly from the reduction semantics [20].

A *translation* of $\mathcal{L}_1 = (\mathcal{P}_1, \longmapsto_1, \asymp_1)$ into $\mathcal{L}_2 = (\mathcal{P}_2, \longmapsto_2, \asymp_2)$, written $[\![\cdot]\!] : \mathcal{L}_1 \rightarrow \mathcal{L}_2$, is a function from $\mathcal{P}_1$ into $\mathcal{P}_2$. We shall call *encoding* any translation that satisfies

the criteria we are going to present. Moreover, to simplify reading, we let S range over processes of the source language (viz., $\mathcal{L}_1$) and T range over processes of the target language (viz., $\mathcal{L}_2$). Notice that, since we aim at a set of criteria suitable for both encodability and separation results, we have to find a compromise between 'minimality' (typical of separation results, where one wants to identify the minimal set of properties that make a separation result provable) and 'maximality' (typical of encodability results, where one wants to show that the encoding satisfies as many properties as possible).

First of all, a translation should be compositional, i.e. the translation of a compound term must be defined in terms of the translation of the subterms, where, in general, the translated subterms can be combined by relying on a context that coordinates their inter-relationships. A *k-ary context* $C[\__1;\ldots;\__k]$ is a term where k occurrences of $\mathbf{0}$ are replaced the holes $\{\__1;\ldots;\__k\}$. In defining compositionality, we let the context used to combine the translated subterms depend on the operator that combines the subterms and on the free names (written $\text{Fn}(\cdot)$) of the subterms. For example, we could think to have a name handler for every free name in the subterms.

Property 1 (Compositionality). A translation $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{L}_2$ is *compositional* if, for every k-ary operator op of $\mathcal{L}_1$ and for every subset of names N, there exists a k-ary context $C^N_{\mathsf{op}}[\__1;\ldots;\__k]$ such that, for all $S_1,\ldots,S_k$ with $\text{Fn}(S_1,\ldots,S_k) = N$, it holds that $[\![\,\mathsf{op}(S_1,\ldots,S_k)\,]\!] = C^N_{\mathsf{op}}[[\![S_1]\!];\ldots;[\![S_k]\!]]$.

Compositionality is a very natural property and, indeed, every encoding we are aware of is defined compositionally. Compositionality with respect to some specific operator has been assumed also to prove some separation result, viz. of synchronous vs asynchronous π-calculus [8] or of persistent fragments of the asynchronous π-calculus [7]. However, for separation results, the most widely accepted criterion is homomorphism of parallel composition [9,17,29,30,32,33]; indeed, translating a parallel process by introducing a coordinating context would reduce the degree of distribution and show that $\mathcal{L}_2$ has not enough expressive power to simulate $\mathcal{L}_1$. This point of view has been, however, sometimes criticized and, indeed, there exist encodings that do not translate parallel composition homomorphically [4,6,26].

Our definition of compositionality allows two processes that only differ in their free names to have totally different translations: indeed, it could be that $C^N_{\mathsf{op}}[\ldots]$ is very different from $C^M_{\mathsf{op}}[\ldots]$, whenever $N \neq M$. We want to avoid this fact; indeed, a "good" translation cannot depend on the particular names involved in the source process, but only on its syntactic structure. However, it is possible that a translation fixes some names to play a precise rôle or it can translate a single name into a tuple of names. Thus, every translation assumes a *renaming policy*, that we now formally define.

Definition 1 (Renaming policy). *Given a translation $[\![\cdot]\!]$, its underlying renaming policy is a function $\varphi_{[\![\,]\!]} : \mathcal{N} \longrightarrow \mathcal{N}^k$, for some constant $k > 0$, such that $\forall u, v \in \mathcal{N}$ with $u \neq v$, it holds that $\varphi_{[\![\,]\!]}(u) \cap \varphi_{[\![\,]\!]}(v) = \emptyset$, where $\varphi_{[\![\,]\!]}(\cdot)$ is simply considered a set here.*

In most of the encodings present in the literature, every name is simply translated to itself. However, it is sometimes necessary to have a set of *reserved* names, i.e. names with a special function within the encoding. Reserved names can be obtained either by

assuming that the target language has more names than the source one, or by exploiting what we call a *strict renaming policy*, i.e. a renaming policy $\varphi_{[\![\,]\!]} : \mathcal{N} \longrightarrow \mathcal{N}$. For example, we can isolate one reserved name by linearly ordering the set of names $\mathcal{N}$ as $\{n_0, n_1, n_2, \ldots\}$ and by letting $\varphi_{[\![\,]\!]}(n_i) \triangleq n_{i+1}$, for every i; the reserved name is n_0.

The requirement that $\varphi_{[\![\,]\!]}$ maps names to tuples of the same length can be justified by the fact that names are all 'at the same level' and, thus, they must be treated uniformly. Moreover, such tuples must be finite, otherwise it would be impossible to transmit all $\varphi_{[\![\,]\!]}(a)$ in the translation of a communication where name a is exchanged (notice that, since the sender cannot know how the receiver will use a, all $\varphi_{[\![\,]\!]}(a)$ must be somehow transmitted). Consequently, the requirement that different names are associated to disjoint tuples can be intuitively justified as follows. Assume that there exists $u \neq v$ such that $\varphi_{[\![\,]\!]}(u) \cap \varphi_{[\![\,]\!]}(v) \neq \emptyset$; since there is no relationship between different names, this implies that, for every w, $\varphi_{[\![\,]\!]}(u) \cap \varphi_{[\![\,]\!]}(w) \neq \emptyset$. If the name shared by every pair of tuples is the same, then such a name can be considered reserved and we can define a renaming policy $\varphi'_{[\![\,]\!]}$ satisfying the requirement of Definition 1. Otherwise, for every v and w, $\varphi_{[\![\,]\!]}(v)$ and $\varphi_{[\![\,]\!]}(w)$ must have a different name in common with $\varphi_{[\![\,]\!]}(u)$; thus, $\varphi_{[\![\,]\!]}(u)$ would contain an infinite number of names.

In our view, a translation should reflect in the translated term all the renamings carried out in the source term. In what follows, we denote with σ a substitution of names for names, i.e. a function $\sigma : \mathcal{N} \longrightarrow \mathcal{N}$, and we shall usually specify only the non-trivial part of a substitution: for example, $\{^b/_a\}$ denotes the (non-injective) substitution that maps a to b and every other name to itself. Moreover, we shall also extend substitutions to tuples of names in the expected way, i.e. component-wise.

Property 2 (Name invariance). A translation $[\![\,\cdot\,]\!] : \mathcal{L}_1 \to \mathcal{L}_2$ is *name invariant* if, for every S and σ, it holds that

$$[\![\,S\sigma\,]\!] \begin{cases} = [\![\,S\,]\!]\sigma' & \text{if } \sigma \text{ is injective} \\ \asymp_2 [\![\,S\,]\!]\sigma' & \text{otherwise} \end{cases}$$

where σ' is such that $\varphi_{[\![\,]\!]}(\sigma(a)) = \sigma'(\varphi_{[\![\,]\!]}(a))$ for every $a \in \mathcal{N}$.

To understand the distinction between injective and non-injective substitutions, assume that σ fuses two (or more) different names. Then, the set of free names of $S\sigma$ is smaller than the set of free names in S; by compositionality, this fact leads to different translations, in general. For example, if the translation introduces a name handler for every free name, having sets of free names with different cardinality leads to inherently different translations. However, non-injective substitutions are natural in name-passing calculi, where language contexts can force name fusions. In this case, the formulation with '=' is too demanding and the weaker formulation (with '$\asymp_2$') is needed. Thus, this formulation implies that two name handlers for the same name are behaviourally equivalent to one handler for that name; this seems to us a very reasonable requirement. Notice that our definition of name invariance is definitely more complex than those, e.g., of [9,29,32,33], where it is required that $[\![\,S\sigma\,]\!] = [\![\,S\,]\!]\theta$ for some (not better specified) substitution θ. However, we do not think that our formulation is more demanding; it is just more detailed and we consider this fact a further contribution of our paper.

Up to now, we have presented and discussed properties dealing with the way in which an translation is defined; we are still left with the more crucial part of the criteria.

We want to focus our attention on the computation capabilities of the languages (i.e., what the languages can calculate); thus, we require that the source and the target language have the same computations. A widely accepted way to formalize this idea is via operational correspondence that, intuitively, ensures two crucial aspects: (*i*) every computation of a source term can be mimicked by its translation (thus, the translation does not reduce the behaviours of the source term); and (*ii*) every computation of a translated term corresponds to some computation of its source term (thus, the translation does not introduce new behaviours).

Property 3 (Operational correspondence). A translation $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{L}_2$ is *operationally corresponding* if it is

Complete: for all $S \Longmapsto_1 S'$, it holds that $[\![S]\!] \Longmapsto_2 \asymp_2 [\![S']\!]$;

Sound: for all $[\![S]\!] \Longmapsto_2 T$, there exists an S' such that $S \Longmapsto_1 S'$ and $T \Longmapsto_2 \asymp_2 [\![S']\!]$.

Notice that operational correspondence is very often used for assessing the quality of an encoding; thus, we have considered it to have a set of criteria that works well both for encodability and for separation results. Nothing related to this property has ever been assumed for separation results, except in [15,16] where, however, it was formulated in a too demanding way. Also notice that the original formulation of operational correspondence put forward in [28] does not use '$\asymp_2$'; for this reason, it is too demanding and, indeed, several encodings (including those in *loc.cit.*) do not enjoy it. The problem is that usually encodings leave some 'junk' process after having mimicked some source language reduction; such a process invalidates the 'exact' formulation of this property. The use of '$\asymp_2$' is justified to get rid of potential irrelevant junks.

Another important semantic issue, borrowed from [9,12,19,26], is that a translation should not introduce infinite computations, written $\longmapsto^{\omega}$.

Property 4 (Divergence reflection). A translation $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{L}_2$ *reflects divergence* if, for every S such that $[\![S]\!] \longmapsto_2^{\omega}$, it holds that $S \longmapsto_1^{\omega}$.

One may argue that divergence can be ignored if it arises with negligible probability or in unfair computations. However, suppose that every translation of $\mathcal{L}_1$ in $\mathcal{L}_2$ introduces some kind of divergence; this means that, to preserve all the functionalities of a terminating source term, every translation has to add infinite computations in the translation of the term. This fact makes $\mathcal{L}_2$ not powerful enough to encode $\mathcal{L}_1$ and is fundamental to proving several separation results (e.g., that the `test-and-set` primitive cannot be encoded via any combination of `read` and `write` – see [19]).

It is interesting to notice that, with all the properties listed up to now, one can accept the translation that maps every source term into $\mathbf{0}$. Of course, this translation is "wrong" because it does not distinguish processes with different interaction capabilities. In process calculi, interaction capabilities are usually described either by the *barbs* that a process exhibits [25] or by the set of *tests* that a process successfully passes [13,34]. Barbs are often defined in a very ad hoc way, are chosen as the simplest predicates that induce meaningful congruences and strictly depend on their language (even though in [34] there is a preliminary attempt at a 'canonical' definition of barbs); for this reason, we found it difficult to work out a satisfactory semantic property relying on

barbs for encodings that translate a source language into a very different target language (notice that barb correspondence is instead very natural in, e.g., [9,17,29] where similar languages are studied). On the contrary, the testing approach is more uniform: it identifies a binary predicate $P \Downarrow O$ of *successful computation* for a process P in a parallel context O (usually called *observer*, that is a normal process containing occurrences of the success term $\surd$), and, by varying O, it describes the interactions P can be engaged in. Moreover, the testing approach is at the same time more general and more elementary than barbs: the latters can be identified via elementary tests and test passing is the basic mechanism for the 'canonical' definition of barbs in [34].

By following [3,7,8], we shall require that the source and the translated term behave in the same way with respect to success. However, a formulation like "$\forall P \forall O.P \Downarrow O$ iff $[\![P]\!] \Downarrow [\![O]\!]$" is not adequate in our setting: indeed, it is possible to have a successful computation for $P|O$ but not for $[\![P]\!] \mid [\![O]\!]$ since, because of compositionality, a successful computation in the target would be possible only with the aid of the coordinating context used to compositionally translate the parallel composition. Thus, we have to define $\Downarrow$ as a unary predicate and require that "$\forall P \forall O.P|O \Downarrow$ iff $[\![P|O]\!] \Downarrow$". For our aims, it is not necessary to distinguish between processes and observers. Moreover, to formulate our property in a simpler way, we assume that all the languages contain the same success process $\surd$ and that $\Downarrow$ means reducibility (in some modality, e.g. may/must/...) to a process containing a top-level unguarded occurrence of $\surd$. This is similar to [17,29], where $\surd$ is an output over a reserved channel and $\Downarrow$ is defined in terms of may and must, respectively. Clearly, different modalities in general lead to different results; in this paper, proof will be carried out in a 'may' modality, but all our results could be adapted to other modalities. Finally, for the sake of coherence, we require the notion of success be caught by the semantic theory underlying the calculi, viz. $\asymp$; in particular, we assume that $\asymp$ never relates two processes P and Q such that $P \Downarrow$ and $Q \not\Downarrow$.

Property 5 (Success sensitiveness). A translation $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{L}_2$ is *success sensitive* if, for every S, it holds that $S \Downarrow$ if and only if $[\![S]\!] \Downarrow$.

3 Proving Known Separation Results

The properties we have just presented are met by most of the best known encodings appearing in the literature (e.g. the encoding of polyadic communications into monadic ones [24], of synchronous into asynchronous communications [5], and so on). Moreover, their combination yields a non-trivial proposal: the first encoding of the asynchronous π-calculus into Mobile Ambients that satisfies all such criteria is in [14]. We now show that their combination allows us to prove in a simpler and more uniform way the best known separation results appearing in the literature.

For example, let us start with the separation results in [17]. There, they work by assuming (a form of) success sensitiveness, homomorphism of '|' and name invariance under any renaming policy that maps every name into a single name. The last two properties, mainly the last one, are debatable. We now prove such results by removing any assumption on the renaming policy and by allowing parallel composition be translated by introducing a centralized coordination process. Thus, we assume that, for every $N \subseteq \mathcal{N}$, there exist $\widetilde{n}$ and R such that $C_{|}^{N}[\_1 \,;\, \_2] = (\nu \widetilde{n})(\_1 \mid \_2 \mid R)$.

Theorem 1. *There exists no encoding of π_a in CCS.*

Proof. By contradiction. Let a, b, c and d be pairwise distinct names and define $P \triangleq [x = b][c = c][d = d]\surd$. Property 3 implies that $[\![(a(x).P \mid \mathbf{0}) \mid \overline{a}\langle b\rangle]\!]$ reduces to a process equivalent to $[\![\surd]\!]$ that, by Property 5, reports success. Let $C_{|}^{\{a,b,c,d\}}[\_1 \,;\, \_2]$ be $(\nu\widetilde{n})(\_1 \mid \_2 \mid R)$; then, $[\![a(x).P \mid \mathbf{0}]\!] \stackrel{\rho}{\Rightarrow} K$ and $[\![\overline{a}\langle b\rangle]\!] \mid R \stackrel{\overline{\rho}}{\Rightarrow} K'$, for $(\nu\widetilde{n})(K \mid K') \asymp_2 [\![\surd]\!]$. In particular, $\rho \triangleq \mu_1 \cdot \ldots \cdot \mu_k$ and $\overline{\rho} \triangleq \overline{\mu}_1 \cdot \ldots \cdot \overline{\mu}_k$, for $\mu_i \in \{m_i, \bar{m}_i\}$ and $\{m_1, \ldots, m_k\} \cap \widetilde{n} = \emptyset$ (indeed, $[\![a(x).P \mid \mathbf{0}]\!] = (\nu\widetilde{n})([\![a(x).P]\!] \mid [\![\mathbf{0}]\!] \mid R)$ and $[\![a(x).P \mid \mathbf{0}]\!] \stackrel{\rho}{\Rightarrow}$ imply that the names in ρ do not belong to $\widetilde{n}$). Let σ be the permutation that swaps a with c and b with d. By Property 2, $[\![c(x).P\sigma \mid \mathbf{0}]\!] \stackrel{\rho'}{\Longrightarrow} K\sigma'$ and $[\![\overline{c}\langle d\rangle]\!] \mid R \stackrel{\overline{\rho'}}{\Longrightarrow} K'\sigma'$, for $(\nu\widetilde{n})(K\sigma' \mid K'\sigma') \asymp_2 [\![\surd]\!]$ and $\rho' = \rho\sigma'$; here σ' denotes the permutation of names induced by σ, as defined in Property 2. More precisely, $\rho' \triangleq \mu'_1 \cdot \ldots \cdot \mu'_k$ and $\overline{\rho'} \triangleq \overline{\mu'_1} \cdot \ldots \cdot \overline{\mu'_k}$, for $\mu'_i \triangleq \mu_i\sigma'$ and $\{\sigma'(m_1), \ldots, \sigma'(m_k)\} \cap \widetilde{n} = \emptyset$.

Now, consider $Q \triangleq ((a(x).P \mid \mathbf{0}) \mid \overline{a}\langle d\rangle) \mid ((c(x).P\sigma \mid \mathbf{0}) \mid \overline{c}\langle b\rangle)$. Trivially, $Q \not\Downarrow$ whereas, as we shall see, $[\![Q]\!] \Downarrow$; this yields the desired contradiction. By compositionality, $[\![Q]\!] \triangleq (\nu\widetilde{n})((\nu\widetilde{n})([\![a(x).P \mid \mathbf{0}]\!] \mid [\![\overline{a}\langle d\rangle]\!] \mid R) \mid (\nu\widetilde{n})([\![c(x).P\sigma \mid \mathbf{0}]\!] \mid ([\![\overline{c}\langle b\rangle]\!] \mid R) \mid R)$. Then, consider $[\![Q]\!] \Longrightarrow (\nu\widetilde{n})((\nu\widetilde{n})(K \mid K') \mid (\nu\widetilde{n})(K\sigma' \mid K'\sigma') \mid R) \asymp_2 (\nu\widetilde{n})([\![\surd]\!] \mid [\![\surd]\!] \mid R)$, obtained by synchronizing

- μ_i produced by $[\![a(x).P]\!]$ with $\overline{\mu}_i$ produced by $[\![\overline{a}\langle d\rangle]\!] \mid R$, if $m_i \notin \varphi_{[\![]\!]}(b)$;
- μ_i produced by $[\![a(x).P]\!]$ with $\overline{\mu'_i}$ produced by $[\![\overline{c}\langle b\rangle]\!] \mid R$, if $m_i \in \varphi_{[\![]\!]}(b)$;
- μ'_i produced by $[\![c(x).P\sigma]\!]$ with $\overline{\mu'_i}$ produced by $[\![\overline{c}\langle b\rangle]\!] \mid R$, if $m_i \notin \varphi_{[\![]\!]}(b)$;
- μ'_i produced by $[\![c(x).P\sigma]\!]$ with $\overline{\mu_i}$ produced by $[\![\overline{a}\langle d\rangle]\!] \mid R$, if $m_i \in \varphi_{[\![]\!]}(b)$. □

Theorem 2. *There exists no encoding of MA in CCS.*

Proof. The previous proof can be adapted to MA: indeed, in [14] we provide an encoding of channel based communications of π_a in MA and in [32] it is shown how to encode name matching in MA. Thus, process $(a(x).[x = b][c = c][d = d]\surd \mid \mathbf{0}) \mid \overline{a}\langle b\rangle$ can be written in MA and the proof then proceeds like above. □

We now aim at proving other separation results, viz. those in [9,29,32,33], in a more uniform and abstract setting. To this aim, however, we must leave the ideal framework presented in Section 2 and make it slightly more concrete; carrying out proofs at the abstract level is a challenging open problem. Mainly, we have to make some assumptions on the semantic theory of the target language, viz. '$\asymp_2$'. We propose three possible instantiations that allow us to develop proofs.

3.1 First Setting

Let us assume that $\asymp_2$ is *exact*, i.e. $T \asymp_2 T'$ and $T \stackrel{\mu}{\rightarrow}$ imply that $T' \stackrel{\mu}{\Rightarrow}$, whenever $\mu \neq \tau$. Notice that examples of exact equivalences are (the different kinds of) synchronous bisimilarity and synchronous trace equivalence. Regretfully, under this assumption, we are able to develop proofs only if $C_{|}^{N}[\_1;\_2]$, the context used to compositionally translate the parallel composition of two processes with free names in N, is $\_1 \mid \_2$, for every set of names N; thus, similarly to [9,17,29,30,32,33], we are now working with encodings that translate '|' homomorphically.

Theorem 3. *Assume that there is a $\mathcal{L}_1$-process S such that $S \not\longmapsto_1$, $S \not\Downarrow$ and $S \mid S \Downarrow$; moreover, assume that every $\mathcal{L}_2$-process T that does not reduce is such that $T \mid T \not\longmapsto_2$. If $\asymp_2$ is exact, then there cannot exist any encoding $[\![\cdot]\!] : \mathcal{L}_1 \longrightarrow \mathcal{L}_2$ that translates '$\mid$' homomorphically.*

Proof. We work by contradiction. First, let us fix, for every $\mathcal{L}_1$-process S that does not reduce, a $\mathcal{L}_2$-process $f([\![S]\!])$ such that $[\![S]\!] \Longrightarrow_2 f([\![S]\!]) \not\longmapsto_2$; such a process always exists because of Property 4 (when $[\![S]\!]$ does not reduce, we can always let $f([\![S]\!]) = [\![S]\!]$). Now, consider the auxiliary encoding $(\!| \cdot |\!) : \mathcal{L}_1 \longrightarrow \mathcal{L}_2$ such that:

$$(\!| S |\!) \triangleq \begin{cases} f([\![S]\!]) & \text{if } S \not\longmapsto_1 \\ (\!| S_1 |\!) \mid (\!| S_2 |\!) & \text{if } S = S_1 \mid S_2 \longmapsto_1 \\ [\![S]\!] & \text{otherwise} \end{cases}$$

Such an encoding satisfies the following two properties:

$$\text{A. if } S \not\longmapsto_1 \text{ then } (\!| S |\!) \not\longmapsto_2 \qquad\qquad \text{B. } (\!| S |\!) \asymp_2 [\![S]\!]$$

Property A follows by construction of $(\!| \cdot |\!)$; let us prove Property B, by induction on the structure of S. If $S \not\longmapsto_1$ (base step and first sub-case of the inductive step), then, by operational completeness (that is part of Property 3), we have that $[\![S]\!] \Longrightarrow_2 f([\![S]\!])$ implies the existence of a S' such that $S \Longrightarrow_1 S'$ and $f([\![S]\!]) \Longrightarrow_2 \asymp_2 [\![S']\!]$. Since $S \not\longmapsto_1$, we have that S' can only be S itself; moreover, the fact that $f([\![S]\!]) \not\longmapsto_1$ implies that $(\!| S |\!) \asymp_2 [\![S]\!]$, as desired. If $S = S_1 \mid S_2 \longmapsto_1$ then, by structural induction, $(\!| S_1 |\!) \asymp_2 [\![S_1]\!]$ and $(\!| S_2 |\!) \asymp_2 [\![S_2]\!]$; we easily conclude by congruence of $\asymp_2$ with respect to parallel composition. The third sub-case is trivial, by reflexivity of $\asymp_2$.

Now, let us take a $\mathcal{L}_1$-process S such that $S \not\longmapsto_1$, $S \not\Downarrow$ and $S \mid S \Downarrow$; by Property 5 and homomorphism, we have that $[\![S]\!] \not\Downarrow$ and $[\![S \mid S]\!] \triangleq [\![S]\!] \mid [\![S]\!] \Downarrow$. This implies that $[\![S]\!] \mid [\![S]\!] \longmapsto_2$, with $[\![S]\!] \xrightarrow{\mu}$ and $[\![S]\!] \xrightarrow{\bar{\mu}}$, for some pair of complementary actions μ and $\bar{\mu}$ (here we are assuming binary synchronizations, as often happens in process calculi). Since $\asymp_2$ is 'exact', we can use property B to obtain that $(\!| S |\!) \xrightarrow{\mu}$ and $(\!| S |\!) \xrightarrow{\bar{\mu}}$; thus, $(\!| S |\!) \mid (\!| S |\!) \longmapsto_2$ whereas, by $S \not\longmapsto_1$ and property A, $(\!| S |\!) \not\longmapsto_2$, in contradiction with the hypothesis. □

Corollary 1. *There exist no encoding of π_{mix}, CCS and MA in π_{sep} that translates '$\mid$' homomorphically.*

Proof. Take any 'exact' behavioural theory for π_{sep} (e.g., strong/branching/weak bisimilarity, both in their early/late/open form, or may/must/fair testing, just to mention some possibilities). On one hand, notice that, if T is a π_{sep}-process such that $T \mid T \longmapsto_2$, then $T \equiv (\nu \widetilde{n})(\Sigma_{i=1}^{m} a_i(x_i).T_i \mid \Sigma_{j=1}^{n} \overline{a'_j}\langle b_j \rangle.T'_j \mid T'')$ and there exist $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$ such that $a_i = a'_j$. Thus, trivially, $T \longmapsto_2$; hence, every π_{sep}-process T that does not reduce is such that $T \mid T \not\longmapsto_2$.

On the other hand, we can find both in CCS, in π_{mix} and in MA a process S that does not reduce and does not report success, but such that $S \mid S$ reports success: it suffices to let S be $a.\mathbf{0} + \bar{a}.\surd$ in CCS, $a(x).\mathbf{0} + \overline{a}\langle b \rangle.\surd$ in π_{mix} and $(\nu p)(open\_p.\surd \mid n[in\_n.p[out\_n.out\_n.\mathbf{0}]])$ in MA. □

3.2 Second Setting

We now consider a second setting where $\asymp_2$, the semantic theory of the target language, is *reduction sensitive*; this means that $T \asymp_2 T'$ and $T' \longmapsto$ imply that $T \longmapsto$. Examples of reduction sensitive equivalence/preorders are strong synchronous/asynchronous bisimulation [1,22] and the expansion preorder [2].

Working under this assumption has the advantage that we are able to carry out proofs also under translations of '|' more liberal than the homomorphic one. As already said, the fact that parallel composition must be translated homomorphically can be criticized and some authors [26] advocate a more liberal formulation, that we now consider. In particular, for every N, we let $C^N_{|}[\__1;\__2] = (\nu\widetilde{n})(\__1 \mid \__2 \mid R)$, for some process R and restricted names $\widetilde{n}$ that only depend on N. We would like to remark that an unconstrained form of compositionality (where nothing is said on $C^N_{|}[\__1;\__2]$) would not change the validity of the results we obtain; it would just force us to prove Theorems 4 and 5 is specific cases and not in a general setting, as now they are.

A Uniform Approach to Separation Results. We now describe the methodological approach we shall follow to prove separation results. The key fact that will enable all our proofs is the following (adapted from [15] and corresponding to property A in the proof of Theorem 3).

Proposition 1. *If $\asymp_2$ is reduction sensitive and $[\![\cdot]\!] : \mathcal{L}_1 \longrightarrow \mathcal{L}_2$ is an encoding, then $S \not\longmapsto_1$ implies that $[\![S]\!] \not\longmapsto_2$, for every S.*

Proof. By contradiction, assume that $[\![S]\!] \longmapsto_2 T$, for some $S \not\longmapsto_1$. By operational soundness, there exists a S' such that $S \Longmapsto_1 S'$ and $T \Longmapsto_2 T' \asymp_2 [\![S']\!]$; but the only such S' is S itself. Since $\asymp_2$ is reduction sensitive and since $[\![S']\!] = [\![S]\!] \longmapsto_2$, then $T' \longmapsto_2 T''$. Again, by operational soundness $T'' \Longmapsto_2 T''' \asymp_2 [\![S]\!]$, and so on; thus, $[\![S]\!] \longmapsto_2 T \longmapsto_2 T'' \longmapsto_2 \ldots$, in contradiction with Property 4 (since $S \not\longmapsto_1$ implies that S does not diverge). □

Another crucial consequence of our criteria is the following proposition.

Proposition 2. *Let $[\![\cdot]\!] : \mathcal{L}_1 \longrightarrow \mathcal{L}_2$ be an encoding and $\asymp_2$ be reduction sensitive. If there exist two $\mathcal{L}_1$-terms S_1 and S_2 such that $S_1 \mid S_2 \Downarrow$, with $S_i \not\Downarrow$ and $S_i \not\longmapsto$ for $i = 1, 2$, then $[\![S_1]\!] \mid [\![S_2]\!] \longmapsto$.*

Proof. In this proof, let us assume the following notation: $block(S)$ denotes any term S' such that $\text{Fn}(S') = \text{Fn}(S)$, $S' \not\longmapsto_1$ and $S' \asymp_1 \mathbf{0}$. It is easy to build such a S': it suffices to prefix S with any blocking action involving a fresh restricted name.

By Properties 1 and 5, $[\![S_1 \mid S_2]\!] = (\nu\widetilde{n})([\![S_1]\!] \mid [\![S_2]\!] \mid R) \Downarrow$. However, since none of $[\![S_1]\!]$, $[\![S_2]\!]$ and $[\![block(S_1) \mid block(S_2)]\!]$ can report success, it must be that $[\![S_1 \mid S_2]\!] \longmapsto_2$. This can only happen either because $[\![S_1]\!] \mid R \longmapsto_2$, or because $[\![S_2]\!] \mid R \longmapsto_2$, or because $[\![S_1]\!] \mid [\![S_2]\!] \longmapsto_2$. The first two possibilities are impossible, because otherwise $[\![S_1 \mid block(S_2)]\!] \longmapsto_2$ or $[\![block(S_1) \mid S_2]\!] \longmapsto_2$ and this would violate Proposition 1: $S_1 \mid block(S_2) \not\longmapsto$ because $S_1 \not\longmapsto_1$, $block(S_2) \not\longmapsto_1$ and $block(S_2) \asymp_1 \mathbf{0}$, and similarly for $block(S_1) \mid S_2$. □

In this framework, the way in which we prove a separation result between $\mathcal{L}_1$ and $\mathcal{L}_2$ is the following:

(a) by contradiction, suppose that there exists an encoding $[\![\cdot]\!] : \mathcal{L}_1 \longrightarrow \mathcal{L}_2$;
(b) find a pair of $\mathcal{L}_1$-processes S_1 and S_2 that satisfy the hypothesis of Proposition 2; by such a result, $[\![S_1]\!] \mid [\![S_2]\!] \longmapsto$;
(c) from S_2 obtain a process S_2' such that $S_1 \mid S_2' \not\longmapsto$ but $[\![S_1]\!] \mid [\![S_2']\!] \longmapsto$;
(d) by Property 1, this implies that $[\![S_1 \mid S_2']\!] \longmapsto$, in contradiction with Proposition 1.

Notice that the identification of S_1 and S_2 (point (b) above) is usually very simple: they are directly obtained from the constructs of $\mathcal{L}_1$ that one believes not to be encodable in $\mathcal{L}_2$. This is different from [9,17,29,32,33] where, instead, a lot of efforts must be spent to define a programming scenario that can be properly implemented in the source language but not in the target one. Point (c) is the only part that requires some ingenuity (it can be easy or quite difficult): usually, it strongly relies on Property 2 (sometimes also on compositionality) to slightly modify S_2 in order to obtain the new process S_2'.

A Simpler Proof of Known Separation Results. First, we reformulate Theorem 3 by changing the hypothesis on $\asymp_2$; this modification will allow us to obtain Corollary 1 under a different choice of semantic theories for π_{sep}.

Theorem 4. *Assume that there is a $\mathcal{L}_1$-process S such that $S \not\longmapsto_1$, $S \not\Downarrow$ and $S \mid S \Downarrow$; moreover, assume that every $\mathcal{L}_2$-process T that does not reduce is such that $T \mid T \not\longmapsto_2$. Also assume that $\asymp_2$ is reduction sensitive. Then, there cannot exist any encoding $[\![\cdot]\!] : \mathcal{L}_1 \longrightarrow \mathcal{L}_2$.*

Proof. By contradiction. Let S be such that $S \not\longmapsto_1$, $S \not\Downarrow$ and $S \mid S \Downarrow$; by Proposition 2, $[\![S]\!] \mid [\![S]\!] \longmapsto_2$ that, by hypothesis, implies that $[\![S]\!] \longmapsto_2$, in contradiction with Proposition 1. □

We now give a second proof-technique that allows us to obtain the hierarchy for polyadic synchronizations in [9] and to adapt the results in [15,16] to the present setting. To this aim, let us define the *matching degree* of a language $\mathcal{L}$, written $\text{MD}(\mathcal{L})$, as the greatest number of names that must be matched to yield a reduction in $\mathcal{L}$. For example, the matching degree of CCS [22], of the π-calculus [22] and of Mobile Ambients [11] is 1; the matching degree of Dπ [18] is 2; the matching degree of π^n (the π-calculus with n-ary polyadic synchronizations [9]) is n; the matching degree of e^π (the π-calculus with arbitrary polyadic synchronizations [9]) is ∞. Indeed, as a representative sample, the π-calculus process $a(x).P \mid \overline{a}\langle b\rangle.Q$ can reduce because of the successful matching between the channel name specified for input and for output (a here).[1]

Theorem 5. *If $\text{MD}(\mathcal{L}_1) > \text{MD}(\mathcal{L}_2)$, then there exists no encoding $[\![\cdot]\!] : \mathcal{L}_1 \longrightarrow \mathcal{L}_2$.*

[1] Incidentally, the early-style LTS for the π-calculus also verifies that $\xrightarrow{\bar{a}b}$ synchronizes with $\xrightarrow{ab}$. However, this does not imply that the matching degree of the π-calculus is 2. Indeed, the process that generates label ab can generate label ac, for every name c; thus, the only name that is matched is the name of the communication channel (a in this case), whereas the second name (viz. b) is only a parameter exchanged.

Proof. By contradiction. Pick two $\mathcal{L}_1$-processes S_1 and S_2 that satisfy the hypothesis of Proposition 2 and that synchronize only once (before reporting success) by matching exactly $k = \text{MD}(\mathcal{L}_1)$ names, viz. $\{n_1, \ldots, n_k\}$. By Proposition 2, their encodings must synchronize: i.e., $[\![S_1]\!] \xrightarrow{\mu}$ and $[\![S_2]\!] \xrightarrow{\bar{\mu}}$. Since $\text{MD}(\mathcal{L}_1) > \text{MD}(\mathcal{L}_2)$, it must be that the names in $\text{FN}(\mu) \cap \text{FN}(\bar{\mu})$ matched when synchronizing μ and $\bar{\mu}$ (say, $\{m_1, \ldots, m_h\}$) are less than k; this implies the existence of an n_i such that $\varphi_{[\![\,]\!]}(n_i) \cap \{m_1, \ldots, m_h\} = \emptyset$. Let us choose a fresh m (i.e., $m \notin \text{FN}(S_1) \cup \text{FN}(S_2) \cup \text{FN}(\mu) \cup \text{FN}(\bar{\mu})$) and consider the substitution σ that swaps m and n_i. Trivially, $S_1 \mid S_2\sigma \mapsto\!\!\!\!\!\!\rightarrow_1$, whereas their encodings do synchronize (in contradiction with Proposition 1): by Property 2, $[\![S_2\sigma]\!] = [\![S_2]\!]\sigma' \xrightarrow{\bar{\mu}\sigma'}$, with $\bar{\mu}\sigma'$ that is still synchronizable with μ because σ' swaps component-wise $\varphi_{[\![\,]\!]}(n_i)$ and $\varphi_{[\![\,]\!]}(m)$ (and so it does not touch $\{m_1, \ldots, m_h\}$). □

Corollary 2. *There exists no encoding from e^π into π^m, for every m, and from π^m into π^n, whenever $m > n$.*

Proof. Observe that $\text{MD}(e^\pi) = \infty$ and that $\text{MD}(\pi^m) = m$; then apply Theorem 5. □

Proving New Separation Results and Building Hierarchies of Languages. We have just shown that our approach is more 'usable' than previous approaches to separation results, since it can be used to prove known results in a simpler and more uniform way. However, it also has the advantage of allowing the proof of new separation results: in [14], we exploit such criteria to compare the relative expressive power of several calculi for mobility (viz., the asynchronous π-calculus, a distributed π-calculus, a distributed version of LINDA, and Mobile/Safe/Boxed Ambients together with several of their variants); moreover, the results in [15,16] can be easily re-formulated under Properties 1–5. Finally, least but not last, the fact that our criteria are also well-suited for encodability results allows us to build hierarchies of languages in a uniform way.

3.3 Third Setting

The setting presented in Section 3.2 relies on the assumption that $\asymp_2$ is reduction sensitive. This restriction seems us not too severe, since most of the operational correspondence results appearing in the literature are formulated up to such semantic theories; the only notable exception we are aware of is [26,28], where weak (asynchronous) bisimilarity [1] is exploited. We now sketch a weaker setting, that covers all the separation results we are aware of (including [26,28]) without breaking the elegant and powerful proof-techniques developed in Section 3.2.

We have said that the aim of formulating operational correspondence up to $\asymp_2$ is to get rid of junk processes possibly arising from the encoding. We can make this intuition explicit by formulating operational correspondence as follows:

- for all $S \Longmapsto_1 S'$, there exist $\widetilde{n}$ and T' such that $[\![S]\!] \Longmapsto_2 (\nu\widetilde{m})([\![S']\!] \mid T') \asymp_2 [\![S']\!]$;
- for all $[\![S]\!] \Longmapsto_2 T$, there exist S', $\widetilde{n}$ and T' such that $S \Longmapsto_1 S'$ and $T \Longmapsto_2 (\nu\widetilde{m})([\![S']\!] \mid T') \asymp_2 [\![S']\!]$.

Maybe, such a formulation can be criticized by saying that it is too 'syntactic', but in practice we are not aware of any encoding that does not satisfy it. Restricting $\asymp_2$

Table 1. Comparison between different separation methodologies. For every result, we list where it appears ('×' if it has never been published and '?' if we believe that it holds but we have not been able to prove it) and the criteria adopted: (*a*) stands for homomorphism w.r.t. '|', (a form of) name invariance and (a form of) success sensitiveness; (*b*) is (*a*) plus a condition requiring that source processes without shared free names must be translated into target processes without shared free names; (*c*) is (*a*) plus divergence reflection.

	Electoral Systems	Matching Systems	Our Criteria		
			1^{st} setting	2^{nd} setting	3^{rd} setting
$CCS \not\rightarrow \pi_{sep}$	[29] (*a*)	×	✓	✓	✓
$\pi_{mix} \not\rightarrow \pi_{sep}$	[29] (*a*)	×	✓	✓	✓
$\mathrm{MA} \not\rightarrow \pi_{sep}$	[32] (*a*)	×	✓	✓	✓
$e^{\pi} \not\rightarrow \pi^{m} \not\rightarrow \pi^{n}$ $(m > n)$	×	[9] (*c*)	?	✓	✓
$\mathrm{MA} \not\rightarrow \mathrm{CCS}$	[33] (*b*)	[17] (*a*)	✓		
$\pi_a \not\rightarrow \mathrm{CCS}$	[29] (*b*)	[17] (*a*)	✓		

to pairs of kind $((\nu\widetilde{n})(T \mid T'), T)$, for $(\nu\widetilde{n})(T \mid T') \asymp_2 T$, yields a reduction sensitive relation, for any $\asymp_2$; thus, Propositions 1 and 2 (and, consequently, all the results proved in Section 3.2) hold also in this setting without requiring reduction sensitiveness of $\asymp_2$.

4 Conclusion

We have presented some criteria that an encoding should satisfy to be considered a good means for language comparison. We have argued that the resulting set of criteria is a satisfactory notion for assessing the relative expressive power of process calculi by noting that most of the best known encodings appearing in the literature satisfy them. Moreover, this notion is not trivial, because there exist known encodings that do not satisfy all the criteria we have proposed: a representative sample is given by the encodings of the π-calculus in Mobile Ambients [10,11].

This paper is mostly methodological, as it describes a new approach both to encodability and to separation results. On one hand, we believe that, for encodability results, we have proposed a valid alternative to full abstraction for comparing languages: our proposal is more focused on expressiveness issues, whereas full abstraction is more appropriate when we look for a tight correspondence between the behavioural equivalences associated with the compared languages. We think that full abstraction is still an interesting notion to investigate when developing an encoding, but it should be considered an "extra-value": if it holds, the encoding is surely more interesting, because it enables not only a comparison of the languages, but also of their associated equivalences. On the other hand, our proposal is also interesting for separation results: as we have shown, several separation results appearing in the literature can be formulated and proved (in an easier and more uniform way) in terms of our criteria. In Table 1 we have comparatively listed such results. Roughly speaking, the approach taken in [9,17,29,32,33] consists in (*i*) identifying a problem that can be solved in the source language but not in the target, and then (*ii*) finding the least set of criteria that an encoding should meet to translate a solution of the problem in the source into a solution of the

problem in the target. Concerning point (*ii*), we have already argued that the criteria put forward by our criteria are not more demanding than those in [9,17,29,32,33]. Concerning point (*i*), we are only aware of two kinds of problem: *symmetric electoral systems* [29,32,33] and *matching systems* [9,17]. However, none of them is 'universal', in the sense that different separation results usually require different separation problems (see the '×' in Table 1).

Of course, there is still a lot of work to do. For example, with the general formulation of our criteria (see Section 2) we have only been able to prove the last two separation results of Table 1, even though we strongly believe that also the remaining ones hold. It would be nice to prove more separation results in the general framework because, in that setting, such results are very strong, being the formulation of our criteria more liberal and abstract.

An orthogonal research line could be the study of enhanced kinds of translation. For example, it may happen to have a 'two-level' encoding [4,6] where $[\![\cdot]\!]$ is a translation that satisfies Properties 2–5 and is such that $[\![P]\!] \triangleq C^{\mathrm{FN}(P)}[(\!|P|\!)]$, where $(\!|\cdot|\!)$ is a compositional translation (this property is called *weak* compositionality in [31]). The proof-techniques presented in Sections 3.2 and 3.3 can be readily adapted to this enhanced notion of encoding, whereas the proof-technique of Section 3.1 cannot (recall that there we had to work with homomorphic translations of parallel composition). Another possibility [21,23] is to define an encoding as a family of translations $[\![\cdot]\!]_{\Xi}$ indexed with a set or a sequence of names Ξ (representing, e.g., an upper bound on the free names of the translated process or some auxiliary parameters for the translation). In this case, our framework is less adequate: it is difficult to adapt our properties and carry out proofs without knowing what the index represents. For example, which is the initial (i.e., top-level) value of Ξ in $[\![\cdot]\!]_{\Xi}$? Are Ξ names in the source or in the target language? The latter question is very delicate: in the first case, Property 2 should be adapted by requiring that $[\![S\sigma]\!]_{\Xi\sigma}$ is equal/equivalent to $([\![S]\!]_{\Xi})\sigma'$; in the second case, we have that $[\![S\sigma]\!]_{\Xi\sigma'}$ must be equal/equivalent to $([\![S]\!]_{\Xi})\sigma'$. Thus, even if we believe that such an enhanced form of encoding is reasonable, we have problems in adapting our framework without specifying anything on the index.

To conclude, the challenge raised in [27] is still open, but we think and hope that our proposal can contribute to its final solution.

Acknowledgments. I am grateful to Daniele Varacca, Jesus Aranda, Frank Valencia and Cosimo Laneve for several comments that improved an earlier draft of this work.

References

1. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous π-calculus. Theoretical Computer Science 195(2), 291–324 (1998)
2. Arun-Kumar, S., Hennessy, M.: An efficiency preorder for processes. Acta Informatica 29(8), 737–760 (1992)
3. Baldamus, M., Parrow, J., Victor, B.: Spi-Calculus Translated to Pi-Calculus Preserving Tests. In: Proc. of LICS, pp. 22–31. IEEE Computer Society, Los Alamitos (2004)
4. Baldamus, M., Parrow, J., Victor, B.: A Fully Abstract Encoding of the Pi-Calculus with Data Terms. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1202–1213. Springer, Heidelberg (2005)

5. Boudol, G.: Asynchrony and the π-calculus (note). Rapp. de Recherche 1702, INRIA (1992)
6. Bugliesi, M., Giunti, M.: Secure implementations of typed channel abstractions. In: Proc. of POPL, pp. 251–262. ACM, New York (2007)
7. Cacciagrano, D., Corradini, F., Aranda, J., Valencia, F.: Persistence and Testing Semantics in the Asynchronous π-calculus. In: Proc. of EXPRESS. ENTCS, vol. 194(2), pp. 59–84 (2007)
8. Cacciagrano, D., Corradini, F., Palamidessi, C.: Separation of Synchronous and Asynchronous Communication Via Testing. Theoretical Computer Science 386(3), 218–235 (2007)
9. Carbone, M., Maffeis, S.: On the expressive power of polyadic synchronisation in pi-calculus. Nordic Journal of Computing 10(2), 70–98 (2003)
10. Cardelli, L., Ghelli, G., Gordon, A.D.: Types for the ambient calculus. Information and Computation 177(2), 160–194 (2002)
11. Cardelli, L., Gordon, A.: Mobile ambients. Theor. Comp. Science 240(1), 177–213 (2000)
12. de Boer, F., Palamidessi, C.: Embedding as a tool for language comparison. Information and Computation 108(1), 128–157 (1994)
13. De Nicola, R., Hennessy, M.: Testing equivalence for processes. TCS 34, 83–133 (1984)
14. Gorla, D.: Comparing calculi for mobility via their relative expressive power. Technical Report 09/2006, Dipartimento di Informatica, Università di Roma La Sapienza
15. Gorla, D.: On the relative expressive power of asynchronous communication primitives. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 47–62. Springer, Heidelberg (2006)
16. Gorla, D.: Synchrony vs asynchrony in communication primitives. In: Proc. of EXPRESS 2006. ENTCS, vol. 175, pp. 87–108. Elsevier, Amsterdam (2007)
17. Haagensen, B., Maffeis, S., Phillips, I.: Matching systems for concurrent calculi. In: Proc. of EXPRESS 2007. ENTCS, vol. 194(2), pp. 85–99 (2007)
18. Hennessy, M., Riely, J.: Resource Access Control in Systems of Mobile Agents. Information and Computation 173, 82–120 (2002)
19. Herlihy, M.: Wait-free synchronization. ACM ToPLaS 13(1), 124–149 (1991)
20. Honda, K., Yoshida, N.: On reduction-based process semantics. TCS 152, 437–486 (1995)
21. Levi, F.: A Typed Encoding of Boxed into Safe Ambients. Acta Inform. 42(6), 429–500 (2006)
22. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
23. Milner, R.: Functions as Processes. Mathem. Struct. in Comp. Science 2(2), 119–141 (1992)
24. Milner, R.: The polyadic π-calculus: A tutorial. In: Logic and Algebra of Specification. Series F. NATO ASI, vol. 94, Springer, Heidelberg (1993)
25. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
26. Nestmann, U.: What is a 'good' encoding of guarded choice? Inf. Comp. 156, 287–319 (2000)
27. Nestmann, U.: Welcome to the jungle: A subjective guide to mobile process calculi. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 52–63. Springer, Heidelberg (2006)
28. Nestmann, U., Pierce, B.C.: Decoding choice encodings. Inf. and Comp. 163, 1–59 (2000)
29. Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous π-calculi. Mathematical Structures in Computer Science 13(5), 685–719 (2003)
30. Palamidessi, C., Saraswat, V., Valencia, F., Victor, B.: On the Expressiveness of Linearity vs Persistence in the Asynchronous π-calculus. In: Proc. of LICS, pp. 59–68. IEEE, Los Alamitos (2006)
31. Parrow, J.: Expressiveness of Process Algebras. In: Proc. of Emerging trends in Concurrency Theory. ENTCS, vol. 209, pp. 173–186. Elsevier, Amsterdam (2008)

32. Phillips, I., Vigliotti, M.: Electoral systems in ambient calculi. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 408–422. Springer, Heidelberg (2004)
33. Phillips, I., Vigliotti, M.: Leader election in rings of ambient processes. Theoretical Computer Science 356(3), 468–494 (2006)
34. Rathke, J., Sassone, V., Sobocinski, P.: Semantic Barbs and Biorthogonality. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 302–316. Springer, Heidelberg (2007)
35. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. C.U.P (2001)

A Notion of Glue Expressiveness for Component-Based Systems

Simon Bliudze and Joseph Sifakis

VERIMAG, Centre Équation, 2 av de Vignate, 38610, Gières, France
{bliudze,sifakis}@imag.fr

Abstract. Comparison between different formalisms and models is often by flattening structure and reducing them to behaviorally equivalent models e.g., automaton and Turing machine. This leads to a notion of expressiveness which is not adequate for component-based systems where separation between behavior and coordination mechanisms is essential. The paper proposes a notion of glue expressiveness for component-based frameworks characterizing their ability to coordinate components.

Glue is a closed under composition set of operators mapping tuples of behavior into behavior. Glue operators preserve behavioral equivalence. They only restrict the behavior of their arguments by performing memoryless coordination.

Behavioral equivalence induces an equivalence on glue operators. We compare expressiveness of two glues G_1 and G_2 by considering whether glue operators of G_1 have equivalent ones in G_2 (strong expressiveness). Weak expressiveness is defined by allowing a finite number of additional behaviors in the arguments of operators of G_2.

We propose an SOS-style definition of glues, where operators are characterized as sets of SOS-rules specifying the transition relation of composite components from the transition relations of their constituents. We provide expressiveness results for the glues of BIP and of process algebras such as CCS, CSP and SCCS. We show that for the considered expressiveness criteria, glues of the considered process calculi are less expressive than general SOS glue. Furthermore, glue of BIP has exactly the same strong expressiveness as glue definable by the SOS characterization.

1 Introduction

A central idea in systems engineering is that complex systems are built by assembling components. Large components are obtained by "gluing" together simpler ones. "Gluing" can be considered as an operation on sets of components.

Component-based techniques have seen significant development, especially through the use of object technologies supported by languages such as C++, Java, and standards such as UML and CORBA. There exist various component frameworks encompassing a large variety of mechanisms for composing components. They focus rather on the way components interact than on their internal behavior. We lack adequate notions of expressiveness to compare the merits and weaknesses of these frameworks.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 508–522, 2008.

Usually, comparison between formalisms and models is by flattening structure and reduction to a behaviorally equivalent model e.g., automaton and Turing machine. In this manner, all finite state formalisms turn out to be expressively equivalent independently of the features used for composition of behaviors. Many models and languages are Turing-expressive, while their coordination capabilities are tremendously different. This fact motivated work on the expressive power of programming languages. Felleisen [1] has provided a framework formally capturing meanings of expressiveness for sequential programming languages and taking into account not only the semantics but also the primitives of languages. Although the general framework is interesting, for component-based systems we need specific results focusing on composition and allowing comparison of different composition operators.

This paper proposes a notion of glue expressiveness for component-based systems. It builds on results and concepts presented in [2] that guided the design of the BIP (Behavior, Interaction, Priority) framework [3]. BIP allows the construction of complex components from atomic ones represented as labeled transition systems, by using two classes of operators: 1) Interaction operators which compose the behavior of their arguments by using interactions (strongly synchronized transitions); 2) Priority operators which are unary operators used to restrict non-determinism of their arguments by disabling an interaction when some interaction of higher priority is enabled.

We consider a framework where composite components are built by application of glue operators. Components are characterized by their behavior and represented in some semantic domain $\mathcal{B}$ equipped with an equivalence relation $\mathcal{R}$. For instance, behaviors can be modeled as labeled transition systems, with an equivalence relation such as strong bisimulation, ready simulation, or simulation. In this case, the behavior of a component consists of all its states and transitions. In general, the behavior is not modified when the component takes a transition. This constitutes an important difference with the process algebra setting, where processes evolve to become other processes.

The concept of glue operator is a generalization of operators in BIP. Parallel composition operators of the process calculi CCS, SCCS, or CSP, are glue operators as well as their unary operators such as labeling, hiding and restriction. Contrary to interleaving, non-deterministic choice is not a glue operator, as it requires additional memory: the choice is applied only once and remains the same for all subsequent transitions of the composed system.

A glue on $\mathcal{B}$ is a closed under composition set G of operators transforming behavior, i.e. mapping tuples of behaviors to behaviors. We require that glue operators only restrict the behavior of their arguments without adding new one, i.e. perform memoryless coordination of behavior.

The equivalence relation $\mathcal{R}$ on $\mathcal{B}$ induces an equivalence relation on glue operators: two n-ary glue operators are equivalent if for any n-tuple of behaviors from $\mathcal{B}$ they give equivalent behaviors. The proposed notion of expressiveness allows the comparison of two glues G_1 and G_2 on the same semantic domain $\mathcal{B}$ by considering whether for any glue operator $gl_1 \in G_1$ there exists an equivalent

operator $gl_2 \in G_2$ (strong expressiveness). Weak expressiveness is defined by allowing a finite number of additional behaviors in the arguments of gl_2.

The main results of the paper can be summarized as follows:

- We propose an SOS-style definition of glues, where operators are characterized as sets of SOS-rules specifying the transition relation of composite components from the transition relations of their constituents. The premises of the rules may involve both positive and negative predicates specifying respectively enabledness or non-enabledness of transitions of components.
- We show that relations, induced on glues by strong bisimulation, ready simulation (preorder and equivalence), and simulation equivalence, coincide. This allows a simple characterization of glue operators as formulae of an algebra generated from the set of the ports of the components by using disjunction, conjunction, and negation operators. The algebra has most of the axioms of a boolean algebra. It does not have the absorption axiom, which is replaced by a weaker one.
- Using this algebraic characterization of glues, we provide expressiveness results for the glues of BIP and of process algebras such as CCS, CSP and SCCS. They show that, for the considered expressiveness criteria, glues of the considered process calculi are less expressive than general glue operators. Furthermore, glue of BIP has exactly the same strong expressiveness as glue definable by the SOS characterization.

The paper is structured as follows. In Sect. 2, we define basic notions that we use in the paper: LTS, SOS-style glue operators, and equivalence relations on these; we provide some results that allow, in particular, to define the algebra of glue formulae, $\mathcal{AGF}(P)$, used to encode the glue operators. In Sect. 3 we introduce the notions of glue expressiveness. In Sect. 4, we use $\mathcal{AGF}(P)$ to compare expressiveness of the glues of CCS, CSP, SCCS, and BIP. We conclude, in Sect. 5, by discussing the results and some directions for future work.

2 Labeled Transition Systems and Glue Operators

In this section, we introduce labeled transition systems (LTS), used to describe behavior, as well as composition operators on these defined in terms of SOS [4].

2.1 Labeled Transition Systems

Definition 1. *A* labeled transition system *is a triple $B = (Q, P, \rightarrow)$, where Q is a set of* states*, P is a set of* ports*, and $\rightarrow \subseteq Q \times 2^P \times Q$ is a set of* transitions*, each labeled by an* action *(i.e. a subset of ports).*

For $q, q' \in Q$ and $a \in 2^P$, we write $q \xrightarrow{a} q'$, if $(q, a, q') \in \rightarrow$. An interaction a is enabled *in state q, denoted $q \xrightarrow{a}$, if there exists $q' \in Q$ such that $q \xrightarrow{a} q'$. If such q' does not exist, a is* disabled*, denoted $q \not\xrightarrow{a}$.*

Notice that reachability related issues are not in the scope of this paper. Consequently, we do not speak of initial states of LTS.

Definition 2. *Let $B_1 = (Q_1, P_1, \rightarrow)$ and $B_2 = (Q_2, P_2, \rightarrow)$ be two LTS, and let $\mathcal{R} \subseteq Q_1 \times Q_2$ be a binary relation. $\mathcal{R}$ is*

1. *a* simulation *iff, for all $q_1 \mathcal{R}\, q_2$, $q_1 \xrightarrow{a} q_1'$ implies $q_2 \xrightarrow{a} q_2'$, for some $q_2' \in Q_2$ such that $q_1' \mathcal{R}\, q_2'$.*
2. *a* ready simulation *iff it is a simulation and, for $q_1 \mathcal{R}\, q_2$, $q_1 \xrightarrow{a}$ implies $q_2 \xrightarrow{a}$.*
3. *a* bisimulation *iff both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations.*

We write $B_1 \sqsubseteq_S B_2$ (resp. $B_1 \sqsubseteq_{RS} B_2$) if there exists a simulation (resp. ready simulation) relating each state of B_1 to some state of B_2. $\sqsubseteq_S$ and $\sqsubseteq_{RS}$ are respectively the simulation *and the* ready simulation preorders *on behaviors. We denote by $\simeq_X = \sqsubseteq_X \cap \sqsubseteq_X^{-1}$, with $X \in \{S, RS\}$, the corresponding equivalences.*

Similarly, $B_1 \leftrightarrows B_2$, iff there exists a bisimulation relating all states of both B_1 and B_2. $\leftrightarrows$ is the bisimulation equivalence *on behaviors.*

Remark 1. It is well known (e.g., [5]) that these relations are connected by the following inclusions: $\leftrightarrows \subseteq \simeq_{RS} \subseteq \simeq_S$ and $\sqsubseteq_{RS} \subseteq \sqsubseteq_S$.

2.2 Glue Operators

Structural Operational Semantics (SOS) [4,6] has been used to define the semantics of programs in terms of LTS. A number of SOS formats have been developed, using various syntactic features [7].

We consider a very simple setting focusing exclusively on behavior composition. In the context of component-based systems, definition of glue only requires the specification of parallel composition operators, as sequential and recursive computation can be represented by individual behaviors. Below, we propose an SOS rules format for component-based composition.

Definition 3. *An n-ary* glue operator *gl is defined as follows. The application of gl to behaviors $B_i = (Q_i, P_i, \rightarrow)$, for $i \in [1, n]$, is a behavior $gl(B_1, \ldots, B_n) = (Q, P, \rightarrow)$, with state space $Q = \prod_{i=1}^{n} Q_i$ the Cartesian product of Q_i, set of ports $P = \bigcup_{i=1}^{n} P_i$, and the maximal transition relation derivable with a set of inference rules of the form*

$$r = \frac{\{B_i : q_i \xrightarrow{a_i} q_i'\}_{i \in I} \qquad \{B_j : q_j \xrightarrow{b_{jk}} \mid k \in [1, m_j]\}_{j \in J}}{gl(B_1, \ldots, B_n) : q_1 \ldots q_n \xrightarrow{a} \widetilde{q_1} \ldots \widetilde{q_n}} \begin{matrix} (premises) \\ (conclusion) \end{matrix}, \quad (1)$$

where $I, J \subseteq [1, n]$; $a = \bigcup_{i \in I} a_i$; and $\widetilde{q_i} = q_i'$, for $i \in I$, and $\widetilde{q_i} = q_i$, for $i \notin I$. Premises of the form $B : q \xrightarrow{a} q'$ are called positive*, those of the form $B : q \xrightarrow{a}$ are called* negative*. Additionally, we require that*

1. *for each $i \in [1, n]$, r has at most one positive premise involving B_i;*
2. *r has at least one positive premise;*
3. *a label can appear either in positive or in negative premises, but not in both.*[1]

[1] A rule with contradictory premises would never be applicable. We include this condition, as it simplifies further proofs and formulations.

We denote by $Pos(r)$ and $Neg(r)$ the sets of positive and negative premises of r respectively (notice that a rule is completely defined by its premises). We identify the glue operator gl with its defining set of derivation rules. A glue operator having no negative premises in any of its derivation rules is called a positive glue operator.

Lemma 1 ([5]). *Glue operators preserve ready simulation and bisimulation, i.e. $B_1 \mathcal{R}\, B_1'$ implies, $gl(B_1, B_2, \ldots, B_n) \mathcal{R}\, gl(B_1', B_2, \ldots, B_n)$, for any behaviors $B_1, \ldots, B_n$, and B_1', an n-ary glue operator gl, and $\mathcal{R} \in \{\sqsubseteq_{RS}, \simeq_{RS}, \underline{\leftrightarrow}\}$.*

The simulation preorder is preserved by positive glue operators.

Example 1 (Rendezvous). Consider the family of binary operators $\rho_{a,b}$, parameterized by two labels. For each pair of labels $a, b \in 2^P$, the composite behavior $\rho_{a,b}(B_1, B_2)$ is inferred from B_1 and B_2 by the set of rules

$$\frac{B_1 : q_1 \xrightarrow{a} q_1' \qquad B_2 : q_2 \xrightarrow{b} q_2'}{\rho_{a,b}(B_1, B_2) : q_1 q_2 \xrightarrow{ab} q_1' q_2'}. \tag{2}$$

and, for all $x \neq a$ and $y \neq b$,

$$\frac{B_1 : q_1 \xrightarrow{x} q_1'}{\rho_{a,b}(B_1, B_2) : q_1 q_2 \xrightarrow{x} q_1' q_2}, \quad \frac{B_2 : q_2 \xrightarrow{y} q_2'}{\rho_{a,b}(B_1, B_2) : q_1 q_2 \xrightarrow{y} q_1 q_2'}. \tag{3}$$

For two behaviors B_1, B_2 having transitions labeled respectively by a and b, $\rho_{a,b}(B_1, B_2)$ is the parallel composition of B_1 and B_2, where a strong synchronization of a and b is the only possible action.

Example 2 (Broadcast). Consider the family of binary operators $\beta_{a,b}$, parameterized by two labels. For each pair of labels $a, b \in 2^P$, the composite behavior $\beta_{a,b}(B_1, B_2)$ is inferred from B_1 and B_2 by the set of rules

$$\frac{B_1 : q_1 \xrightarrow{a} q_1'}{\beta_{a,b}(B_1, B_2) : q_1 q_2 \xrightarrow{a} q_1' q_2}, \quad \frac{B_1 : q_1 \xrightarrow{a} q_1' \qquad B_2 : q_2 \xrightarrow{b} q_2'}{\beta_{a,b}(B_1, B_2) : q_1 q_2 \xrightarrow{ab} q_1' q_2'}. \tag{4}$$

For two behaviors B_1, B_2 having transitions labeled respectively by a and b, $\beta_{a,b}(B_1, B_2)$ is the parallel composition of B_1 and B_2, where interactions a and b are weakly synchronized with a being the trigger. In other words, B_2 can perform a transition on b only if it is synchronized with a transition of B_1 on a.

Example 3 (Priority). Consider the family of unary operators $\pi_{a,b}$, parameterized by two labels. For each pair of labels $a, b \in 2^P$, the composite behavior $\pi_{a,b}(B)$ is inferred from B by the set of rules

$$\frac{B : q \xrightarrow{a} q' \qquad B : q \not\xrightarrow{b}}{\pi_{a,b}(B) : q \xrightarrow{a} q'}, \quad \frac{q \xrightarrow{b} q'}{\pi_{a,b}(B) : q \xrightarrow{b} q'}. \tag{5}$$

For a behavior B having transitions labeled by a and b, $\pi_{a,b}(B)$ is the restriction of B, where the interaction a can only happen if b is not possible, i.e. a has lower priority than b.

2.3 Relations on Glue Operators

The relations on LTS defined above are canonically extended to glue operators.

Definition 4. *For $\mathcal{R} \in \{\sqsubseteq_S, \sqsubseteq_{RS}, \simeq_S, \simeq_{RS}, \underline{\leftrightarrow}\}$, the relation $\mathcal{R}$ is extended to glue operators by putting, for any two n-ary glue operators gl_1 and gl_2,*

$$gl_1 \mathcal{R}\, gl_2 \stackrel{def}{\Longleftrightarrow} \forall B_1, \ldots, B_n,\ gl_1(B_1, \ldots, B_n) \mathcal{R}\, gl_2(B_1, \ldots, B_n)\,. \tag{6}$$

Clearly, the inclusions of Rem. 1 also hold for relations on glue operators.

Lemma 2. *Two glue operators $gl_1 = \{r_1\}$ and $gl_2 = \{r_1, r_2\}$, with $Pos(r_1) = Pos(r_2)$ and $Neg(r_1) \subseteq Neg(r_2)$, are bisimilar $gl_1 \underline{\leftrightarrow} gl_2$.*

Proof. The proof follows immediately from the definition of bisimilarity. It is based on the fact that, whenever r_2 is applicable, r_1 can also be applied. □

Definition 5. *If a glue operator does not have redundant rules as in Lem. 2, we say that it is* without redundancy*.*

Lemma 3. *Let gl_1, gl_2 be glue operators, and gl_1 be without redundancy. $gl_1 \sqsubseteq_S gl_2$ implies that, for each rule $r_1 \in gl_1$, there is a rule $r_2 \in gl_2$ having $Pos(r_2) = Pos(r_1)$ and $Neg(r_2) \subseteq Neg(r_1)$.*

Proof. Consider the rule (cf. Def. 3)

$$r_1 \;=\; \frac{\{B_i : q_i \xrightarrow{a_i} q_i'\}_{i \in I} \qquad \{B_j : q_j \stackrel{b_{j_k}}{\not\longrightarrow} \mid k \in [1, m_j]\}_{j \in J}}{gl(B_1, \ldots, B_n) : q_1 \ldots q_n \xrightarrow{a} \widetilde{q_1} \ldots \widetilde{q_n}} \;\in gl_1\,,$$

and, for $i \in [1, n]$, $B_i^1 = (Q_i, P, \rightarrow_i)$ having $Q_i = \{q^i\}$ and $\rightarrow_i$ defined by

$$\rightarrow_i \;= \begin{cases} \{q^i \xrightarrow{a} q^i \mid a \in 2^P\}, & \text{for } i \notin J, \\ \{q^i \xrightarrow{a} q^i \mid a \in 2^P\} \setminus \{q^i \xrightarrow{b_{i_k}} q^i \mid k \in [1, m_i]\}, & \text{for } i \in J\,. \end{cases} \tag{7}$$

Both behaviors obtained by applying gl_1 and gl_2 to $B_1^1, \ldots, B_n^1$ have exactly one state that we denote respectively q' and q''.

All the premises of r_1 are satisfied in q'. Hence $q' \xrightarrow{a} q'$ in $gl_1(B_1^1, \ldots, B_n^1)$. By simulation $gl_1 \sqsubseteq_S gl_2$, we also have $gl_1(B_1^1, \ldots, B_n^1) \sqsubseteq_S gl_2(B_1^1, \ldots, B_n^1)$. Hence, $q'' \xrightarrow{a} q''$ in $gl_2(B_1^1, \ldots, B_n^1)$, and there exists a rule $r_2 \in gl_2$ enabling this transition. Thus, $Pos(r_2) = Pos(r_1)$ and $Neg(r_2) \subseteq Neg(r_1)$. □

Proposition 1. *Let gl_1, gl_2 be glue operators without redundancy. Then $gl_1 \simeq_S gl_2$ implies $gl_1 = gl_2$, where $=$ is the equality of sets of derivation rules.*

Proof. Consider a rule $r_1 \in gl_1$. By Lem. 3, $gl_1 \sqsubseteq_S gl_2$ implies that there exists $r_2 \in gl_2$ having $Pos(r_2) = Pos(r_1)$ and $Neg(r_2) \subseteq Neg(r_1)$, whereas $gl_2 \sqsubseteq_S gl_1$ implies that there exists $r_1' \in gl_1$ having $Pos(r_1') = Pos(r_2)$ and $Neg(r_1') \subseteq Neg(r_2)$. By non-redundancy of gl_1, we conclude $r_1' = r_1 = r_2$, i.e. $gl_1 \subseteq gl_2$. By symmetry, this proves the proposition. □

Proposition 2. *Let gl_1, gl_2 be glue operators without redundancy. Then $gl_1 \sqsubseteq_{RS} gl_2$ implies $gl_1 = gl_2$, where $=$ is the equality of sets of derivation rules.*

Proof. 1) Let $gl_1 \sqsubseteq_{RS} gl_2$ and consider a rule $r_1 \in gl_1$. By Lem.3, there exists $r_2 \in gl_2$, such that $Pos(r_2) = Pos(r_1)$ and $Neg(r_2) \subseteq Neg(r_1)$. Suppose that $Neg(r_2) \subsetneq Neg(r_1)$, i.e. there exists a negative premise $(B : q \stackrel{b}{\not\rightarrow}) \in Neg(r_1) \setminus Neg(r_2)$. Consider, for $i \in [1, n]$, the behaviors $B_i^2 = (Q_i, P, \rightarrow_i)$, constructed as in (7), but removing the transition corresponding to this premise. As in the proof of Lem.3, we denote q' and q'' the unique states of $gl_1(B_1^2, \ldots, B_n^2)$ and $gl_2(B_1^2, \ldots, B_n^2)$ respectively. Clearly all the premises of r_2 are still satisfied and a transition $q'' \stackrel{a}{\rightarrow} q''$ is possible in $gl_2(B_1^2, \ldots, B_n^2)$. On the other hand, the premises of r_1 are no longer satisfied.

Suppose that there exists another rule $r_1' \in gl_1$, which allows the transition $q' \stackrel{a}{\rightarrow} q'$ in $gl_1(B_1^2, \ldots, B_n^2)$. As above, we have $Pos(r_1') = Pos(r_1) = Pos(r_2)$ and $Neg(r_1') \subseteq Neg(r_2) \subseteq Neg(r_1)$, which violates the non-redundancy assumption.

Assuming that there is no such rule $r_1' \in gl_1$, we conclude that $q' \stackrel{a}{\not\rightarrow}$ in $gl_1(B_1^2, \ldots, B_n^2)$, which, by ready simulation, implies a contradiction: $q'' \stackrel{a}{\not\rightarrow}$ in $gl_2(B_1^2, \ldots, B_n^2)$. Hence, $Neg(r_2) = Neg(r_1)$, i.e. $r_1 = r_2 \in gl_2$ and $gl_1 \subseteq gl_2$.

2) Assume now that $gl_1 \subsetneq gl_2$, i.e. there exists a rule $r_2 \in gl_2 \setminus gl_1$ with conclusion labeled by some interaction a. We consider again the behaviors B_i^1, for $i \in [1, n]$, constructed as above. Again, we have $q'' \stackrel{a}{\rightarrow} q''$ in $gl_2(B_1^1, \ldots, B_n^1)$, and, by ready simulation, $q' \stackrel{a}{\rightarrow} q'$ in $gl_1(B_1^1, \ldots, B_n^1)$. Hence, there exists $r_1 \in gl_1 \subseteq gl_2$ enabling this transition. As above, we have $Pos(r_1) = Pos(r_2)$ and $Neg(r_1) \subseteq Neg(r_2)$, which contradicts the non-redundancy of r_2. □

This proves the following theorem, implying that to compare glue operators it is sufficient to compare the corresponding sets of SOS rules.

Theorem 1. *Bisimulation, ready simulation preorder and equivalence, and simulation equivalence on glue operators coincide: $\underline{\leftrightarrow} = \simeq_{RS} = \simeq_S = \sqsubseteq_{RS}$. All these relations coincide with the equality of operators as sets of rules.*

2.4 The Algebra of Glue Formulae

Theorem 1 also allows to define an algebraic encoding of glue operators, which we use, in particular, to define the composition of glue operators. Glue composition must preserve essential information about atomic behavior. For instance, if interaction a is inhibited by some other interaction b, this relation must be maintained even when, in the composed system, b cannot be fired for some other reason: b must be synchronized with another interaction that is not enabled in the current state, or it is itself inhibited by another interaction.

For instance, assume that firing the interaction a takes one of the components to a critical state, for which mutual exclusion must be ensured, whereas firing b takes another component out of such state. If b is possible, a should not be fired (as this would violate the mutual exclusion) even if b is inhibited by another interaction c.

Although, a definition of composition, which respects these requirements, can be given directly in terms of glue operators, it is much simpler and more intuitive to give it in terms of the algebra presented below. An up to bisimulation one-to-one correspondence between formulae of the algebra and glue operators ensures the translation of composition back to glue operators.

Syntax. Let P be a set of ports, such that $0, 1 \notin P$. The syntax of the algebra of glue formulae, $f \in \mathcal{AGF}(P)$, is given by

$$\begin{array}{l} f ::= f \vee f \mid f \wedge t \mid e\,, \\ t ::= (t \vee t) \mid \neg e \mid e\,, \\ e ::= e \vee e \mid e \wedge e \mid (e) \mid a \in 2^P \mid 0 \mid 1\,, \end{array} \tag{8}$$

where the three operations, denoted by '$\neg$', '$\wedge$', and '$\vee$' are respectively unary *negation* and binary *conjunction* and *disjunction* (in order of decreasing binding power). We often omit '$\wedge$' and represent conjunction by simple juxtaposition.

Intuitively, e represents a positive expression, whereas t is a term which can have a negation at top level, i.e. the negated term must be purely positive. As t can only appear in conjunction with another term, a negative term, in $\mathcal{AGF}(P)$ formulae, is always in conjunction with a positive term.

Axioms. Both $\wedge$ and $\vee$ are associative, commutative, idempotent, and distribute over each other; 0 is the unit for $\vee$; 1 is the unit for $\wedge$; $f \wedge 0 = 0$. Negation satisfies all the axioms except double negation and excluded middle:

1. $\neg 0 = 1$ and $\neg 1 = 0$,
2. $f \wedge \neg f = 0$,
3. $\neg f_1 \wedge \neg f_2 = \neg(f_1 \vee f_2)$,
4. $\neg f_1 \vee \neg f_2 = \neg(f_1 \wedge f_2)$.

Lemma 4 (Restricted absorption). $\forall f_1, f_2 \in \mathcal{AGF}(P)$, $f_1 \neg f_2 \vee f_1 = f_1$.

Lemma 5. *Each formula $f \in \mathcal{AGF}(P)$ has a disjunctive normal form, i.e. a representation as a disjunction of conjunctions of positive and negative variables.*

Proof (Sketch). This follows from the existence of the DNF of boolean formulae and the fact that the syntax (8) of $\mathcal{AGF}(P)$ guarantees that negation only appears at the top level. This property is also preserved by de Morgan's laws. □

Semantics. The semantics of $\mathcal{AGF}(P)$ is given in terms of glue operators. It depends on the mapping of ports in P to components. For a system with n atomic components $B_1, \ldots, B_n$, the partition of P is given by a function $\kappa : P \to [1, n]$, such that a port $p \in P$ belongs to the component $B_{\kappa(p)}$. The function κ trivially extends to interactions within one component.

The meaning of a clause $a_1 \ldots a_k \wedge \neg b_1 \ldots \neg b_l$ is given by the rule r, having

$$\begin{array}{l} Pos(r) = \left\{ B_{\kappa(a_i)} : q_{\kappa(a_i)} \xrightarrow{a_i} q'_{\kappa(a_i)} \;\middle|\; i \in [1,k] \right\}, \\ Neg(r) = \left\{ B_{\kappa(b_i)} : q_{\kappa(b_i)} \not\xrightarrow{b_i} \;\middle|\; i \in [1,l] \right\}. \end{array}$$

Clearly, the DNF of $f \neq 0, 1$, does not contain occurrences of neither 0 nor 1. The meaning of the formula $f \neq 0, 1$ is then given by the glue operator gl_f defined

by the set of rules, corresponding to clauses of the DNF of f. The meaning of 0 is the operator defined by the empty set of derivation rules, which blocks all interactions of all the components in the system.

Proposition 3. *The axiomatization of $\mathcal{AGF}(P)$ is sound and complete, i.e., for two formulae $f_1, f_2 \in \mathcal{AGF}(P)$, $f_1 = f_2$ if and only if $gl_{f_1} \leftrightarrows gl_{f_2}$.*

For any glue operator gl, there exists $f \in \mathcal{AGF}(P)$, such that $gl \leftrightarrows gl_f$.

Proof (Sketch). Clearly, the semantic construction above is one-to-one between $\mathcal{AGF}(P)$ formulae and glue operators without redundancy. The second part follows directly from Lemmas 2 and 4. □

This proposition allows to identify the glue operators with their corresponding glue formulae. We use this to define a composition of glue operators. The usual composition is not compatible with the restriction that all interactions in positive premises of a rule must participate in the conclusion (cf. Def.3).

Example 4. Consider two operators defined by the corresponding formulae $f = a\neg b \vee b \vee c$ and $g = a \vee b\neg c \vee c$ (cf. also the opening of this section). The usual composition $f \circ g$ consists here in substituting $b\neg c$ for b in f. Thus, in f, $a\neg b$ becomes $a\neg(b\neg c) = a\neg b \vee a\neg\neg c$, b becomes $b\neg c$, and c stays the same. This gives $f \circ g = a\neg b \vee a\neg\neg c \vee b\neg c \vee c$. However, $a\neg\neg c$ is not authorized by the syntax of $\mathcal{AGF}(P)$. In terms of SOS, this would correspond to having a positive premise c that would not participate in the conclusion of the rule.

Consider two glue operators defined by the formulae $f = \bigvee_{i\in I} a_i x_i$ and $g = \bigvee_{j\in J} b_j y_j$, where, for $i \in I$ and $j \in J$, a_i and b_j are conjunctions of positive interaction variables, whereas x_i and y_j are purely negative expressions.

Definition 6. *The composition of f with g is defined by*

$$f * g \stackrel{def}{=} \bigvee_{i\in I} \bigvee_{K\subseteq J} \left(x_i \wedge \bigwedge_{k\in K} b_k y_k \right) = \bigvee_{i\in I} \bigvee_{K\subseteq J} \left(a_i x_i \wedge \bigwedge_{k\in K} y_k \right), \qquad (9)$$

where the inner disjunction is taken over all $K \subseteq J$, such that $\bigwedge_{k\in K} b_k = a_i$.

Example 5. Taking on the previous example, we have $f * g = a\neg b \vee b\neg c \vee c$. Thus, when all three interactions a, b, and c are ready to be fired, both a and b are inhibited by b and c respectively.

3 Expressiveness of Glue

Let $\mathcal{B}$ be a set of behaviors with a fixed equivalence relation $\mathcal{R} \subseteq \mathcal{B} \times \mathcal{B}$. A glue is a set G of operators on $\mathcal{B}$. We denote by $\mathcal{G}lue$ the set of all glues on $\mathcal{B}$. We denote $G^{(n)} \subseteq G$ the set of all n-ary operators in G. Thus, $G = \bigcup_{n\geq 1} G^{(n)}$.

To determine whether one glue is more expressive than another, we compare their respective sets of behaviors *composable* from the same *atomic* ones. Several

approaches to comparing the expressiveness of glues can be considered according to the type of modifications of the system that one allows in order to perform the comparison. In any case, this consists in exhibiting for each operator of one glue an equivalent operator in the other one. Below, we define two criteria for the comparison of glue expressiveness:

1. *Strong* expressiveness, where the exhibited glue operator must be applied to the same set of behaviors as the original one,
2. *Weak* expressiveness, where the exhibited glue operator must be applied to the same set of behaviors as the original one, with potentially an addition of some fixed set of *coordination behaviors*.

Definition 7. *For a given set $\mathcal{B}$ and an equivalence $\mathcal{R}$ on $\mathcal{B}$, the* strong *expressiveness preorder $\preccurlyeq_S \subseteq \mathcal{G}lue \times \mathcal{G}lue$ w.r.t. $\mathcal{R}$ is defined by putting, for G_1, $G_2 \in \mathcal{G}lue$, $G_1 \preccurlyeq_S G_2$ if, for any $n \geq 1$ and $B_1, \ldots, B_n \in \mathcal{B}$,*

$$\forall gl_1 \in G_1^{(n)} \ \exists gl_2 \in G_2^{(n)} : gl_1(B_1, \ldots, B_n) \,\mathcal{R}\, gl_2(B_1, \ldots, B_n)\,. \tag{10}$$

Definition 8. *For a given set $\mathcal{B}$ and an equivalence $\mathcal{R}$ on $\mathcal{B}$, the* weak *expressiveness preorder $\preccurlyeq_W \subseteq \mathcal{G}lue \times \mathcal{G}lue$ w.r.t. $\mathcal{R}$ is defined by putting, for G_1, $G_2 \in \mathcal{G}lue$, $G_1 \preccurlyeq_W G_2$ if there exists a finite subset $\mathcal{C} \subset \mathcal{B}$ of coordination behaviors such that, for any $n \geq 1$ and $B_1, \ldots, B_n \in \mathcal{B}$,*

$$\begin{aligned} \forall gl_1 \in G_1^{(n)} \ \exists C_1, \ldots, C_m \in \mathcal{C}, gl_2 \in G_2^{(n+m)} : \\ gl_1(B_1, \ldots, B_n) \,\mathcal{R}\, gl_2(B_1, \ldots, B_n, C_1, \ldots, C_m)\,. \end{aligned} \tag{11}$$

These two preorders allow to define the following notions for glue comparison.

Definition 9. *Let $G_1, G_2 \in \mathcal{G}lue$. The following relations are defined w.r.t. $\mathcal{R}$.*

1. *G_1 and G_2 are* strongly equivalent *if $G_1 \preccurlyeq_S G_2$ and $G_2 \preccurlyeq_S G_1$.*
2. *G_1 and G_2 are* weakly equivalent *if $G_1 \preccurlyeq_W G_2$ and $G_2 \preccurlyeq_W G_1$.*
3. *G_1 is* strongly more expressive *than G_2 if $G_2 \preccurlyeq_S G_1$ and $G_1 \not\preccurlyeq_W G_2$.*
4. *G_1 is* weakly more expressive *than G_2 if $G_2 \preccurlyeq_W G_1$ and $G_1 \not\preccurlyeq_W G_2$.*

Remark 2. The two order relations ("strongly more expressive" and "weakly more expressive") defined above are partial orders (as opposed to preorders). However, notice that we define the relation "strongly more expressive" to be stronger than the canonical order induced by the preorder $\preccurlyeq_S$. As $G_1 \preccurlyeq_S G_2$ implies $G_1 \preccurlyeq_W G_2$, the case $G_1 \preccurlyeq_S G_2$ and $G_2 \preccurlyeq_W G_1$ fits the case 2 above, i.e. G_1 and G_2 are weakly equivalent.

Example 6. We consider behaviors to be LTS. Let $\mathcal{P}$ be a universal set of ports. We define two glues Bin and Ter generated respectively by families of binary and ternary rendezvous operators: $\rho^{(2)}_{a,b}$ and $\rho^{(3)}_{a,b,c}$ for all $a, b, c \in 2^{\mathcal{P}}$ (cf. Ex. 1).

Clearly, $Ter \preccurlyeq_S Bin$, as for any $a, b \in 2^{\mathcal{P}}$, and any B_1, B_2, B_3, we have $\rho^{(3)}_{a,b,c}(B_1, B_2, B_3) = \rho^{(2)}_{a,bc}\Big(B_1, \rho^{(2)}_{b,c}(B_2, B_3)\Big)$. On the contrary, $Bin \not\preccurlyeq_W Ter$, as any two components at any given state can only perform two actions (one action each), whereas three are needed for a ternary synchronization. We conclude that Bin is strongly more expressive than Ter.

We have supposed so far that systems are built from components with disjoint sets of ports and that all actions are observable. To compare expressiveness of formalisms that do not meet this requirements, we adapt the definition of glue expressiveness by using labeling functions that modify the labeling of transitions without otherwise affecting the transition relation.

Definition 10. *Let $\mathcal{B}$ be a set of behaviors with a universal set of ports P and $\mathcal{R}$ an equivalence on $\mathcal{B}$. Let $\varphi, \psi : P \to P$ be two given labeling functions, such that $\psi \circ \varphi = id$. The strong (φ, ψ)-expressiveness preorder $\preccurlyeq_S^{(\varphi,\psi)} \subseteq \mathcal{G}lue \times \mathcal{G}lue$ w.r.t. $\mathcal{R}$ is defined by putting, for G_1, $G_2 \in \mathcal{G}lue$, $G_1 \preccurlyeq_S^{(\varphi,\psi)} G_2$ iff, for any $n \geq 1$ and $B_1, \ldots, B_n \in \mathcal{B}$,*

$$\forall gl_1 \in G_1^{(n)} \; \exists gl_2 \in G_2^{(n)} : gl_1(B_1, \ldots, B_n) \mathcal{R}\, \psi\left(gl_2\Big(\varphi(B_1), \ldots, \varphi(B_n)\Big)\right), \quad (12)$$

where e.g., $\varphi(B)$ is the behavior obtained from B by applying φ to labels of all the transitions in B.

Definitions 8 and 9 are adapted analogously.

Example 7. Taking on the previous example, consider $\tau \notin \mathcal{P}$, and let $C = (\{1\}, \{\tau\}, \to)$ be an LTS with the only transition $1 \xrightarrow{\tau} 1$.

For any B_1, B_2 and $a, b \in 2^{\mathcal{P}}$, we have $\rho^{(2)}_{a,b}(B_1, B_2)\, \mathcal{R}\, \psi\Big(\rho^{(3)}_{a,b,\tau}(B_1, B_2, C)\Big)$, where ψ is a labeling function erasing all occurrences of τ, and $\mathcal{R}$ is any of the equivalence relations discussed in Sect. 2.3. Thus, $Bin \preccurlyeq_W^{(id,\psi)} Ter$, i.e. Bin and Ter are weakly (id, ψ)-equivalent.

4 Glues of BIP and Process Algebras

In the following sections, we compare the glues of BIP [3] and those of classical calculi: CSP [8], CCS [9], and SCCS [10]. All these glues are positive and consist in their respective parallel composition and restriction operators.

Lemma 6. *Let G_1 and G_2 be two positive glues (i.e. consisting of only positive glue operators). $G_1 \preccurlyeq_S G_2$ with respect to any of $\simeq_S$, $\simeq_{RS}$, and $\underline{\leftrightarrow}$ iff $G_1 \subseteq G_2$.*

Proof (Sketch). Consider a family of behaviors, each having one state with loop transitions on all corresponding interactions. To enable exactly the same transitions, two positive glue operators must have exactly the same rules. □

4.1 BIP

In BIP [3], behavior composition is by means of interaction models – sets of interactions, described by connectors [11] – and priority models (partial orders on interactions), used to enforce scheduling policies applied to interactions.

The composition operator, defined by a set of interactions $\gamma \subseteq 2^P$, is given by the $\mathcal{AGF}(P)$ formula $int_\gamma = \bigvee \gamma$ (the disjunction of all the interactions in γ).

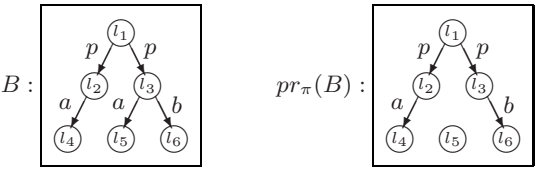


Fig. 1. Example behavior for the proof of Prop. 5

We denote by IM the set of all such glue operators. As interactions are sets of ports, operators in IM are purely positive (each clause of such an operator being a conjunction of positive port variables).

A *priority model* π is a strict partial order on 2^P. For $a, a' \in 2^P$, we write $a \prec a'$ iff $(a, a') \in \pi$, meaning that interaction a has less priority than a'. The priority operator is given by the $\mathcal{AGF}(P)$ formula

$$pr_\pi \;=\; \bigvee_{a \in 2^P} \left(a \wedge \bigwedge_{a \prec a'} \neg a' \right) .$$

We denote BIP the glue consisting of all operators obtained by composition (cf. Def. 6) of interaction and priority operators.

Proposition 4. *IM is strongly equivalent to the set of all positive glue operators, whereas BIP is strongly equivalent to the set of all glue operators.*

Proof (Sketch). The first affirmation is trivial. It is also clear from the above presentation that any operator in BIP is a glue operator in the sense of Def. 3. To show that any glue operator can be realized in BIP, we represent it as a DNF formula $f \in \mathcal{AGF}(P)$. Each conjunctive clause of f has a positive and a negative part. The positive parts of all clauses uniquely define the the set of interactions γ, whereas regrouping (by de Morgan's laws) the negative parts of all clauses with the same positive part defines the priority model. □

Proposition 5. *BIP is strongly more expressive than IM w.r.t. $\simeq_S$ (*a fortiori *$\simeq_{RS}$ and $\underline{\leftrightarrow}$).*

Proof (Sketch). First of all, $IM \subset BIP$ and IM contains only positive operators. Hence, by Lem. 6, $IM \preccurlyeq_S BIP$ and $BIP \not\preccurlyeq_S IM$. As, in BIP, all interactions are visible this also implies $BIP \not\preccurlyeq_W IM$. It is easy to show $BIP \not\preccurlyeq_W^{(id,\psi)} IM$, with ψ erasing all the ports of coordination behavior, by considering the priority model $\pi = \{(a, b)\}$ applied to the behavior B (see Fig. 1). □

4.2 CCS and SCCS

In both CCS and SCCS [9,10], one considers the set A of actions along with the set $\overline{A} = \{\overline{a} \,|\, a \in A\}$ of complementary actions and the non-observable action τ. $L = A \cup \overline{A} \cup \{\tau\}$ is the set of labels.

To render the action sets of different components disjoint, we consider (φ, ψ)-expressiveness. We define $\varphi(B)$ to be a behavior obtained from B by renaming any action $a \in A \cup \overline{A}$ of B to $B.a$. Conversely $\psi(B)$ renames any action $B.a$ of B to a. Furthermore, for any behaviors B_1, B_2 we put $\psi(B_1.a\, B_2.\overline{a}) = \tau$.

The glue of CCS consists of operators obtained by hierarchical composition of the parallel composition and restriction operators. Parallel composition $\|$ operator allows binary synchronization of complementary actions $a, \overline{a} \in L$. Restriction $\backslash a$ excludes a given action $a \in A$ and its complement $\overline{a}$ from communication, thus enforcing synchronization $a\,\overline{a}$, when it is possible.

For a system composed of n atomic behaviors $B_1, \ldots, B_n$, we consider prefixed labels as ports, i.e. $P = \{B_i.a \,|\, i \in [1, n], a \in L\}$. The CCS parallel composition operator is expressed in $\mathcal{AGF}(P)$ by the formula

$$par_{CCS} = \bigvee_{a \in A} \bigvee_{i,j=1}^{n} B_i.a\, B_j.\overline{a} \quad \vee \quad \bigvee_{a \in A} \bigvee_{i=1}^{n} (B_i.a \vee B_i.\overline{a} \vee B_i.\tau)\,. \tag{13}$$

The unary restriction (i.e. applied to a single component) and the n-ary restriction (i.e. combined with the parallel composition of n components) operators are given respectively by

$$rst_{a,1} = \bigvee_{i=1}^{n} \bigvee_{l \neq a, \overline{a}} B_i.l\,, \tag{14}$$

$$rst_{a,n} = \bigvee_{l \in A} \bigvee_{i,j=1}^{n} B_i.l\, B_j.\overline{l} \quad \vee \quad \bigvee_{l \in A \setminus \{a\}} \bigvee_{i=1}^{n} (B_i.l \vee B_i.\overline{l} \vee B_i.\tau)\,. \tag{15}$$

All operators in the CCS glue are positive. Moreover, a conjunctive clause of the corresponding $\mathcal{AGF}(P)$ formula consists of at most two ports. As the labeling ψ can only erase ports, this remains true even in presence of coordination behavior. These observations allow us to conclude that, with φ and ψ as above, IM is strongly more (φ, ψ)-expressive than the CCS glue.

In SCCS, labels are elements of a free Abelian group Act generated by A (with $\overline{a}$ being the inverse of a). The glue of SCCS also consists of hierarchical combinations of the parallel composition and restriction operators. Parallel composition $\times$ forces all components to synchronize. It is given by the formula

$$par_{SCCS} = \bigwedge_{i=1}^{n} \left(B_i.\tau \vee \bigvee_{a \in A} B_i.a \right)\,. \tag{16}$$

Restriction operator in SCCS is complementary to that of CCS, i.e. it states the actions that are visible, rather than those that are invisible. Although, it can be easily defined in terms of $\mathcal{AGF}(P)$, we omit this definition here.

The SCCS glue is also positive, which, as above, allows to conclude that it is strongly less (φ, ψ)-expressive than IM. The opposite relation remains to be investigated. The fact that conjunctive clauses of $SCCS$ operators can comprise more than two ports suggests that CCS glue is not weakly more expressive than SCCS glue.

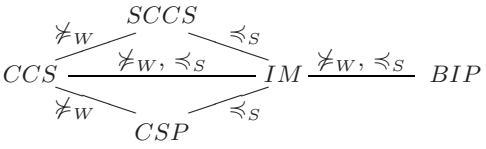


Fig. 2. Summary of relations between glues

4.3 CSP

In CSP [8], processes communicate over a set C of *channels* common to the system. Again, we consider the same relabeling functions φ and ψ as in the previous sections, and ports $P = \{B_i.c \,|\, i \in [1,n], c \in C \cup \{\tau\}\}$.

Again the glue of CSP consists of hierarchical combinations of the parallel composition and restriction operator. Parallel composition $\|_{C'}$ is parameterized by the subset $C' \subseteq C$ of channels. Interactions on the channels in C' must synchronize, whereas interactions on other channels interleave. This is given by the formula

$$par_{CSP} = \bigvee_{c \in C'} \bigwedge_{i=1}^{n} B_i.c \;\vee\; \bigvee_{c \notin C'} \bigvee_{i=1}^{n} (B_i.\tau \vee B_i.c)\,. \tag{17}$$

Again, for the sake of brevity, we omit the restriction operator.

The CSP glue is also positive. It can be observed that conjunctive clauses of the corresponding $\mathcal{AGF}(P)$ formulae consist exclusively of ports corresponding to the same channel. This suggests that, as for the CCS glue, IM is strongly more (φ,ψ)-expressive than the CSP glue. Again, the fact that conjunctive clauses of the operators of the CSP can comprise more than two ports suggests that the CCS glue is not weakly more expressive than the CSP glue.

Relations between the glues considered above are summarized in Fig. 2.

5 Conclusion

We studied notions for comparing expressiveness of glues in component-based frameworks. In contrast to usual notions, they enforce separation between behavior and composition operators. For instance, it is not possible to have as in process algebras, expansion theorems expressing parallel composition in terms of non-deterministic choice and prefixing by actions.

The definition of glue operators considers transitions of composite components as the result of the transitions of their constituents. We showed that bisimilarity, ready simulation (preorder and equivalence), and simulation equivalence coincide, when canonically extended to glue operators. This allows the characterization of glues as formulae, which drastically simplifies the comparison and composition of glues.

We have not yet completely explored possible relations between glues of process calculi. However, they cannot be as expressive as glues with negative

premises and this weakness cannot be overcome even by allowing additional behavior for coordination.

We have kept the framework as simple as possible. We only consider behavioral preorders where all the ports are observable. The robustness of the presented results for expressiveness based on observational relations should be investigated. Furthermore, we have not considered rules with lookahead premises (e.g., [7]) which seems to increase expressiveness of positive rules.

We proposed a framework for dealing with expressiveness of composition operators. This is a step towards breaking with reductionistic approaches which consider glue operators only as behavior transformers. It allows setting up criteria for comparing component-based languages and understanding their strengths and weaknesses.

Acknowledgements

The authors are grateful to Philippe Bidinger, Yassine Lakhnech, and the anonymous reviewers for valuable discussion and constructive comments regarding this paper.

References

1. Felleisen, M.: On the expressive power of programming languages. In: Jones, N.D. (ed.) ESOP 1990. LNCS, vol. 432, pp. 134–151. Springer, Heidelberg (1990)
2. Sifakis, J.: A framework for component-based construction. In: 3rd IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM 2005), September 2005, pp. 293–300 (2005) (Keynote talk)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: 4th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM 2006), September 2006, pp. 3–12 (2006) (Invited talk)
4. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19. University of Aarhus (1981)
5. Bloom, B.: Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages. PhD thesis, Massachusetts Institute of Technology (1989)
6. Aceto, L., Fokkink, W., Verhoef, C.: Structural Operational Semantics. In: Handbook of Process Algebra, ch.3, pp. 197–292. Elsevier, Amsterdam (2001)
7. Mousavi, M., Reniers, M.A., Groote, J.F.: SOS formats and meta-theory: 20 years after. Theoretical Computer Science 373(3), 238–272 (2007)
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1985)
9. Milner, R.: Communication and Concurrency. Prentice Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1989)
10. Milner, R.: Calculi for synchrony and asynchrony. Theoretical Computer Science 25(3), 267–310 (1983)
11. Bliudze, S., Sifakis, J.: The algebra of connectors — Structuring interaction in BIP. In: Proc. of the EMSOFT 2007, October 2007, pp. 11–20. ACM SigBED (2007)

Author Index