LNCS 3921

Luca Aceto
Anna Ingólfsdóttir (Eds.)

Foundations of Software Science and Computation Structures

9th International Conference, FOSSACS 2006
Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006
Vienna, Austria, March 2006, Proceedings

Lecture Notes in Computer Science 3921

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

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
New York University, NY, USA

Doug Tygar
University of California, Berkeley, CA, USA

Moshe Y. Vardi
Rice University, Houston, TX, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Luca Aceto   Anna Ingólfsdóttir (Eds.)

Foundations of Software Science and Computation Structures

9th International Conference, FOSSACS 2006
Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006
Vienna, Austria, March 25-31, 2006
Proceedings

Springer

Volume Editors

Luca Aceto
Anna Ingólfsdóttir

Reykjavík University,
Department of Computer Science
Ofanleiti 2, 103 Reykjavík, Iceland
E-mail:{luca,annai}@ru.is

Library of Congress Control Number: 2006922023

CR Subject Classification (1998): F.3, F.4.2, F.1.1, D.3.3-4, D.2.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-33045-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-33045-5 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2006
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11690634 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2006 was the ninth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 18 satellite workshops (ACCAT, AVIS, CMCS, COCV, DCC, EAAI, FESCA, FRCSS, GT-VMT, LDTA, MBT, QAPL, SC, SLAP, SPIN, TERMGRAPH, WITS and WRLA), two tutorials, and seven invited lectures (not including those that were specific to the satellite events). We received over 550 submissions to the five conferences this year, giving an overall acceptance rate of 23%, with acceptance rates below 30% for each conference. Congratulations to all the authors who made it to the final programme! I hope that most of the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate Program Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2006 was organized by the Vienna University of Technology, in cooperation with:

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);
- European Association of Software Science and Technology (EASST);
- Institute for Computer Languages, Vienna;
- Austrian Computing Society;
- The *Bürgermeister der Bundeshauptstadt Wien*;
- Vienna Convention Bureau;
- Intel.

The organizing team comprised:

Chair:	Jens Knoop
Local Arrangements:	Anton Ertl
Publicity:	Joost-Pieter Katoen
Satellite Events:	Andreas Krall
Industrial Liaison:	Eva Kühn
Liaison with City of Vienna:	Ulrich Neumerkel
Tutorials Chair, Website:	Franz Puntigam
Website:	Fabian Schmied
Local Organization, Workshops Proceedings:	Markus Schordan

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Perdita Stevens (Edinburgh, Chair), Luca Aceto (Aalborg and Reykjavík), Rastislav Bodík (Berkeley), Maura Cerioli (Genova), Matt Dwyer (Nebraska), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Roberto Gorrieri (Bologna), Reiko Heckel (Leicester), Michael Huth (London), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Shriram Krishnamurthi (Brown), Kim Larsen (Aalborg), Tiziana Margaria (Göttingen), Ugo Montanari (Pisa), Rocco de Nicola (Florence), Hanne Riis Nielson (Copenhagen), Jens Palsberg (UCLA), Mooly Sagiv (Tel-Aviv), João Saraiva (Minho), Don Sannella (Edinburgh), Vladimiro Sassone (Southampton), Helmut Seidl (Munich), Peter Sestoft (Copenhagen), Andreas Zeller (Saarbrücken).

I would like to express my sincere gratitude to all of these people and organizations, the Program Committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, the many reviewers, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the Organizing Chair of ETAPS 2006, Jens Knoop, for arranging for us to have ETAPS in the beautiful city of Vienna.

Edinburgh
January 2006

Perdita Stevens
ETAPS Steering Committee Chair

Preface

This volume collects the proceedings of "Foundations of Software Science and Computation Structures," FOSSACS 2006. FOSSACS is a member conference of ETAPS, the "European Joint Conferences on Theory and Practice of Software," dedicated to foundational research for software science. It invites submissions on theories and methods to underpin the analysis, integration, synthesis, transformation, and verification of programs and software systems. Topics covered usually include: algebraic models; automata and language theory; behavioral equivalences; categorical models; computation processes over discrete and continuous data; computation structures; logics of programs; modal, spatial, and temporal logics; models of concurrent, reactive, distributed, and mobile systems; models of security and trust; language-based security; process algebras and calculi; semantics of programming languages; software specification and refinement; type systems and type theory.

FOSSACS 2006 consisted of one invited and 28 contributed papers, selected out of 107 submissions, yielding an acceptance rate of roughly 26%. The quality of the submitted papers was very high indeed, and several good manuscripts could not be selected for presentation at the conference by the Program Committee. This indicates that FOSSACS is by now an established conference on theoretical computer science to which the authors are submitting some of their best work.

Besides the contributed papers, this volume includes an article by Wan Fokkink, the FOSSACS invited speaker. Wan's contribution, entitled '*On Finite Alphabets and Infinite Bases II: Completed and Ready Simulation*,' is coauthored with Taolue Chen and Sumit Nain—two young, up-and-coming researchers—and presents new results on the equational theory of simulation-based preordering relations between concurrent processes.

The order of presentation of the contributed papers in this volume follows the structure of the program for the conference.

We owe a huge debt of gratitude to the Program Committee for their sterling effort during the difficult process of selecting a program for the conference; to the referees, for carrying out the reviewing tasks with outstanding competence, care, and timeliness; and ultimately to the authors for making our selection very hard by submitting their best work to FOSSACS. Thanks to Jens Knoop for the local organization, and to Martin Karusseit for his support with the conference electronic management system.

We hope that you will enjoy reading this volume.

Reykjavík
January 2006

Luca Aceto and Anna Ingólfsdóttir
Program Chairs
FOSSACS 2006

Organization

Program Committee

Luca Aceto (Reykjavík, Iceland)
Roberto Amadio (Paris VII, France)
Bruno Blanchet (ENS Paris, France)
Gerard Boudol (INRIA Sophia Antipolis, France)
Nadia Busi (Bologna, Italy)
Luca Cardelli (Microsoft Research, UK)
Flavio Corradini (Camerino, Italy)
Luca de Alfaro (Santa Cruz, USA)
Zoltan Ésik (Szeged, Hungary)
Thomas Henzinger (EPFL, Switzerland)
Anna Ingólfsdóttir (Reykjavík, Iceland)
Bengt Jonsson (Uppsala , Sweden)
Dexter Kozen (Cornell, USA)
Antonin Kucera (Brno, Czech Republic)
Orna Kupferman (Jerusalem, Israel)
Marta Kwiatkowska (Birmingham, UK)
Catuscia Palamidessi (INRIA/Futurs, France)
Erik Poll (Nijmegen, The Netherlands)
Alban Ponse (Amsterdam, The Netherlands)
Edmund Robinson (Queen Mary College, UK)
Vladimiro Sassone (Sussex, UK)
Steve Schneider (Surrey, UK)
Igor Walukiewicz (Labri, France)
Thomas Wilke (Kiel, Germany)

Referees

Rezine Ahmed
Carlos Areces
Vincent Balat
Michael Baldamus
Paolo Baldan
Vince Barany
Franco Barbanera
Massimo Bartoletti
Emmanuel Beffara
Nick Benton
Joshua Berdine
Martin Berger
Rudolf Berghammer
Marco Bernardo
Yves Bertot
Dietmar Berwanger
Inge Bethke
Dirk Beyer
Karthik Bhargavan
Gavin Bierman
Andreas Blass
Stefan Blom
Achim Blumensath
Frank de Boer
Johannes Borgstroem
Julian Bradfield
Tomas Bradzil
Mario Bravetti
Diletta R. Cacciagrano
Cristiano Calcagno
Marco Carbone
Ilaria Castellani
Dario Catalano
Rohit Chadha
Krishnendu Chatterjee
Alessandra Cherubini
Yannick Chevalier
Tom Chothia
Corina Cirstea

Giovanni Conforti
Byron Cook
Martin Cooper
Andrea Corradini
Veronique Cortier
Jean-Michel Couvreur
Silvia Crafa
Rosario Culmone
Frederic Dabrowski
Silvano Dal Zilio
Giorgio Delzanno
Jolie de Miranda
Moshe Deutsch
Razvan Diaconescu
Maria Rita Di Berardini
Bob Diertens
Pietro Di Gianantonio
Marie Duflot-Kremer
Jan van Eijck
Neil Evans
Maribel Fernandes
Riccardo Focardi
Cedric Fournet
Adrian Francalanza
Sibylle Froeschle
Maurizio Gabbrielli
Fabio Gadducci
David Galindo
Blaise Genest
Georges Gonthier
Andrew D. Gordon
Clemens Grabmayer
Stefano Guerrini
Christian Haack
Magnus M. Halldórsson
James Heather
Frederic Herbreteau
Thomas Hildebrandt
Thai Son Hoang
Jan Holecek
Engelbert Hubbers
Marieke Huisman
Hans Hüttel
Samuel Hym
Lucian Ilie
Bart Jacobs
Radha Jagadeesan
Petr Jancar
David Janin
Ole H. Jensen
Thierry Joly
Christine Julien
Jarkko Kari
Felix Klaedtke
Bartek Klin
Naoki Kobayashi
Simon Kramer
Tomas Kratochvila
Steve Kremer
Hans-Joerg Kreowski
Ralf Kuesters
Alexander Kurz
Anna Labella
Cosimo Laneve
Martin Lange
James Leifer
Daniel Leivant
Giacomo Lenzi
Jerome Leroux
Martin Leucker
Jean-Jacques Levy
Paul Levy
Huimin Lin
Etienne Lozes
Denis Lugiez
Yoad Lustig
Bas Luttik
Carsten Lutz
Parthasarathy Madhusudan
Henning Makholm
Claude Marche
Ralph Matthes
Guy McCusker
Alistair McEwan
Paul-Andre Mellies
Emanuela Merelli
Massimo Merro
Marino Miculan
Dale Miller
Anders Moeller
Eugenio Moggi
Sotiris Moschoyiannis
Larry Moss
Wojciech Mostowski
MohammadReza Mousavi
Andrzej Murawski
Anca Muscholl
Gopalan Nadathur
Damian Niwinski
Gethin Norman
Peter O'Hearn
Martijn Oostdijk
Vincent van Oostrom
Friederich Otto
Luca Padovani
David Parker
Augusto Parma
Joachim Parrow
Dirk Pattison
Romain Pechoux
Giovanni Michele Pinna
Adolfo Piperno
Nir Piterman
David Pitt
Francois Pottier
Damien Pous
John Power
Rosario Pugliese
Femke van Raamsdonk
Anders P. Ravn
Vojtech Rehak
Michel Reniers
Eike Ritter
Piet Rodenburg
Michael Rusinowitch
Peter Ryan
Claudio Sacerdoti Coen
Mayank Saksena
Davide Sangiorgi
Alan Schmitt
Roberto Segala
Olivier Serre
Mike Shields
Alex Simpson
Christian Skalka

Jeremy Sproston
Jiří Srba
Oldrich Strazovsky
Jan Strejcek
Grégoire Sutre
Andrzej Tarlecki
David Teller
Luca Tesei
Hendrik Tews
Sophie Tison
Nikola Trcka
Helen Treharne
Mathieu Turuani
Sandor Vagvolgyi
Frank D. Valencia
Vasco T. Vasconcelos
Gerard Verfaillie
Björn Victor
Maria Grazia Vigliotti
Aymeric Vincent
Fer-Jan de Vries
Wang Xu
Daria Walukiewicz
Andrzej Wasowski
Muck van Weerdenburg
Graham White
Kidane Yemane
Tsai Yih-Kuen
Mark van der Zwaag

Table of Contents

Invited Talk

On Finite Alphabets and Infinite Bases II: Completed and Ready Simulation
Taolue Chen, Wan Fokkink, Sumit Nain ... 1

Mobile Processes

A Theory for Observational Fault Tolerance
Adrian Francalanza, Matthew Hennessy ... 16

Smooth Orchestrators
Cosimo Laneve, Luca Padovani ... 32

On the Relative Expressive Power of Asynchronous Communication Primitives
Daniele Gorla ... 47

More on Bisimulations for Higher Order π-Calculus
Zining Cao ... 63

Software Science

Register Allocation After Classical SSA Elimination is NP-Complete
Fernando Magno Quintão Pereira, Jens Palsberg ... 79

A Logic of Reachable Patterns in Linked Data-Structures
Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, Ahmed Bouajjani ... 94

Distributed Computation

Dynamic Policy Discovery with Remote Attestation
Corin Pitcher, James Riely ... 111

Distributed Unfolding of Petri Nets
Paolo Baldan, Stefan Haar, Barbara König ... 126

On the μ-Calculus Augmented with Sabotage
Philipp Rohde 142

Categorical Models

A Finite Model Construction for Coalgebraic Modal Logic
Lutz Schröder 157

Presenting Functors by Operations and Equations
Marcello M. Bonsangue, Alexander Kurz 172

Bigraphical Models of Context-Aware Systems
L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, H. Niss 187

Processes for Adhesive Rewriting Systems
Paolo Baldan, Andrea Corradini, Tobias Heindel, Barbara König, Paweł Sobociński 202

Real Time and Hybrid Systems

On Metric Temporal Logic and Faulty Turing Machines
Joël Ouaknine, James Worrell 217

Denotational Semantics of Hybrid Automata
Abbas Edalat, Dirk Pattinson 231

Process Calculi

Reversing Algebraic Process Calculi
Iain Phillips, Irek Ulidowski 246

Conjunction on Processes: Full–Abstraction Via Ready–Tree Semantics
Gerald Lüttgen, Walter Vogler 261

Undecidability Results for Bisimilarity on Prefix Rewrite Systems
Petr Jančar, Jiří Srba 277

Automata and Logic

Propositional Dynamic Logic with Recursive Programs
Christof Löding, Olivier Serre 292

A Semantic Approach to Interpolation
Andrei Popescu, Traian Florin Şerbănuţă, Grigore Roşu 307

First-Order and Counting Theories of ω-Automatic Structures
Dietrich Kuske, Markus Lohrey 322

Parity Games Played on Transition Graphs of One-Counter Processes
Olivier Serre 337

Domains, Lambda Calculus, Types

Bidomains and Full Abstraction for Countable Nondeterminism
James Laird 352

An Operational Characterization of Strong Normalization
Luca Paolini, Elaine Pimentel, Simona Ronchi Della Rocca 367

On the Confluence of λ-Calculus with Conditional Rewriting
Frédéric Blanqui, Claude Kirchner, Colin Riba 382

Security

Guessing Attacks and the Computational Soundness of Static Equivalence
Martín Abadi, Mathieu Baudet, Bogdan Warinschi 398

Handling exp, × (and Timestamps) in Protocol Analysis
Roberto Zunino, Pierpaolo Degano 413

Symbolic and Cryptographic Analysis of the Secure WS-ReliableMessaging Scenario
Michael Backes, Sebastian Mödersheim, Birgit Pfitzmann, Luca Viganò 428

Author Index 447

On Finite Alphabets and Infinite Bases II: Completed and Ready Simulation*

Taolue Chen[1,2], Wan Fokkink[1,3], and Sumit Nain[4]

[1] CWI, Department of Software Engineering, PO Box 94079, 1090 GB Amsterdam, The Netherlands
chen@cwi.nl
[2] Nanjing University, State Key Laboratory of Novel Software Technology, Nanjing, Jiangsu, P.R. China, 210093
[3] Vrije Universiteit Amsterdam, Department of Theoretical Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
wanf@cs.vu.nl
[4] Rice University, Department of Computer Science, 6100 S. Main Street, Houston, TX 77005-1892, USA
sumitnain@yahoo.com

Abstract. We prove that the equational theory of the process algebra BCCSP modulo completed simulation equivalence does not have a finite basis. Furhermore, we prove that with a finite alphabet of actions, the equational theory of BCCSP modulo ready simulation equivalence does not have a finite basis. In contrast, with an infinite alphabet, the latter equational theory does have a finite basis.

1 Introduction

Labeled transition systems constitute a fundamental model of concurrent computation which is widely used in light of its flexibility and applicability. They model processes by explicitly describing their states and their transitions from state to state, together with the actions that produce them. Several notions of behavioral equivalence have been proposed, with the aim to identify those states of labeled transition systems that afford the same observations. The lack of consensus on what constitutes an appropriate notion of observable behavior for reactive systems has led to a large number of proposals for behavioral equivalences for concurrent processes.

Van Glabbeek [6] presented the linear time - branching time spectrum of behavioral preorders and equivalences for finitely branching, concrete, sequential processes. In this paper we focus on two semantics in this spectrum. A relation R between processes is a *simulation* if $s_0\ R\ s_1$ and $s_0 \xrightarrow{a} s_0'$ implies $s_1 \xrightarrow{a} s_1'$ with $s_0'\ R\ s_1'$. Such a relation is a *completed simulation* if whenever s_0 cannot perform any transition, the same holds for s_1. It is a *ready simulation* if s_0 and s_1 can

* Partially supported by the Dutch Bsik project BRICKS (Basic Research in Informatics for Creating the Knowledge Society), 973 Program of China (No. 2002CB312002), and NNSFC (No. 60233010, No. 60273034, No. 60403014).

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 1–15, 2006.
© Springer-Verlag Berlin Heidelberg 2006

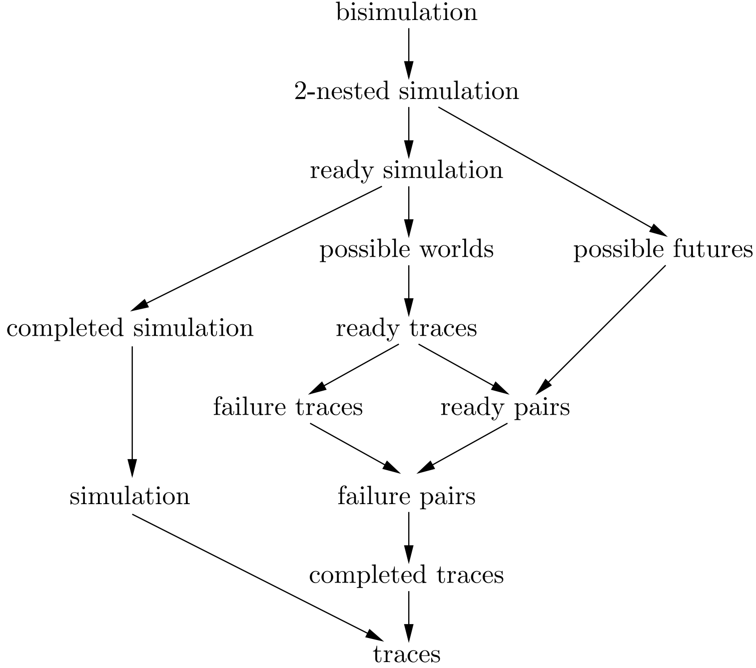


Fig. 1. The linear time - branching time spectrum

perform exactly the same initial actions. Simulation semantics is coarser than completed simulation semantics (meaning that it distinguishes fewer processes), which in turn is coarser than ready simulation semantics. Other semantics in the linear time - branching time spectrum are also based on simulation notions, or on decorated traces. Figure 1 depicts the linear time - branching time spectrum, where a directed edge from one equivalence to another means that the source of the edge is finer than the target.

Van Glabbeek [6] studied the semantics in his spectrum in the setting of the process algebra BCCSP, which contains only basic process algebraic operators from CCS and CSP, but is sufficiently powerful to express all finite synchronization trees. Van Glabbeek gave axiomatizations for the semantics in the spectrum, such that two closed BCCSP terms can be equated by the axioms if and only if they are equivalent.

Having defined a model of an axiomatization for a process algebra in terms of LTSs, it is natural to study the connection between the equations that are valid in the chosen model, and those that are derivable from the axioms using the rules of equational logic. A key question here is whether there is a finite axiomatization that is ω-complete. That is, if all closed instances of an equation can be derived, does this imply that the equation itself can be derived from the axiomatization using the rules of equational logic? (We also refer to an ω-complete axiom system as a *basis* for the algebra it axiomatizes.) An ω-complete axiomatization of a

behavioral congruence yields a purely syntactic characterization, independent of LTSs and of the actual details of the definition of the chosen behavioral equivalence, of the semantics of the process algebra. This bridge between syntax and semantics plays an important role in both the practice and the theory of process algebras. From the point of view of practice, these proof systems can be used to perform system verifications in a purely syntactic way using general purpose theorem provers or proof checkers, and form the basis of purpose-built axiomatic verification tools like, e.g., PAM [8].

A notable example of an ω-incomplete axiomatization in the literature is the equational theory of CCS [9]. Therefore laws such as commutativity of parallelism, which are valid in the initial model but which cannot be derived, are often added to the latter equational theory. For such extended equational theories, ω-completeness results were presented in the setting of CCS [10] and ACP [3].

A number of positive and negative results regarding finite ω-complete axiomatizations for BCCSP occur in the literature. Moller [10] proved that the finite axiomatization for BCCSP modulo bisimulation equivalence is ω-complete. Groote [7] presented a similar result for completed trace equivalence, for trace equivalence (in case of an alphabet with more than one element), and for readiness and failures equivalence (in case of an infinite alphabet). Fokkink and Nain [5] obtained a finite ω-complete axiomatization for BCCSP modulo failures equivalence in case of a finite alphabet, by adding one extra axiom that uses the cardinality of the alphabet. In [4] they proved that in case of a finite alphabet of at least two elements, BCCSP modulo any semantics in between readiness and possible worlds equivalence does not have a finite basis. Blom, Fokkink and Nain [2] proved that in case of an infinite alphabet, BCCSP modulo ready trace equivalence does not have a finite sound and ground-complete axiomatization. Aceto, Fokkink, van Glabbeek and Ingolfsdottir [1] proved a similar negative result for 2-nested simulation and possible futures equivalence, independent of the cardinality of the alphabet.[1]

In this paper we consider BCCSP modulo completed simulation and ready simulation semantics. We prove that no finite sound and ground-complete axiomatization for BCCSP modulo completed simulation preorder and equivalence is ω-complete. To be more precise, we prove that the infinite family of inequations

$$a^n x \preccurlyeq a^n \mathbf{0} + a^n(x+y) \qquad (n \geq 1)$$

which are sound modulo completed simulation preorder, cannot be axiomatized in a finite fashion. This result is surprising in the sense that completed simulation is the only semantics in the linear time - branching time spectrum that in case of an infinite alphabet has a finite sound and ground-complete axiomatization for BCCSP, but no finite ω-complete axiomatization.

[1] In case of an infinite alphabet, occurrences of action names in axioms should be interpreted as variables, as else most of the axiomatizations mentioned in this paragraph would be infinite.

Next we prove that in case of a finite alphabet $\{b_1, \ldots, b_k\}$, no finite sound and ground-complete axiomatization for BCCSP modulo ready simulation preorder and equivalence is ω-complete. To be more precise, we prove that the infinite family of inequations

$$a^n x \preccurlyeq a^n \mathbf{0} + a^n(x + b_1\mathbf{0}) + \cdots + a^n(x + b_k\mathbf{0}) \qquad (n \geq 1)$$

which are sound modulo ready simulation preorder, cannot be axiomatized in a finite fashion.

Finally, we prove, using the technique of inverted substitutions from [7], that in case of an infinite alphabet, the equational theory of BCCSP modulo ready simulation equivalence does have a finite basis.

This paper is set up as follows. Section 2 presents basic definitions regarding simulation semantics, the process algebra BCCSP, and (in)equational logic. Section 3 contains the proofs of the negative results for completed simulation preorder and equivalence. And Section 4 contains the proofs of the negative and positive results for ready simulation preorder and equivalence.

2 Preliminaries

Simulation semantics: A *labeled transition system* contains a set of states, with typical element s, and a set of transitions $s \xrightarrow{a} s'$, where a ranges over some set A of labels. The set $\mathcal{I}(s)$ consists of those $a \in A$ for which there exists a transition $s \xrightarrow{a} s'$.

Definition 1 (Simulation). *Assume a labeled transition system.*

- *A binary relation R on states is a* simulation *if $s_0 \; R \; s_1$ and $s_0 \xrightarrow{a} s_0'$ imply $s_1 \xrightarrow{a} s_1'$ with $s_0' \; R \; s_1'$.*
- *A simulation R is a* completed simulation *if $s_0 \; R \; s_1$ and $\mathcal{I}(s_0) = \emptyset$ imply $\mathcal{I}(s_1) = \emptyset$.*
- *A simulation R is a* ready simulation *if $s_0 \; R \; s_1$ and $a \notin \mathcal{I}(s_0)$ imply $a \notin \mathcal{I}(s_1)$.*

We write $s_0 \precsim_{\mathrm{CS}} s_1$ or $s_0 \precsim_{RS} s_1$ if $s_0 \; R \; s_1$ with R a completed or ready simulation, respectively. The kernels of $\precsim_{\mathrm{CS}}$ and $\precsim_{\mathrm{RS}}$ are denoted by $\simeq_{\mathrm{CS}}$ and $\simeq_{\mathrm{RS}}$, respectively.

Syntax of BCCSP: BCCSP(A) is a basic process algebra for expressing finite process behavior. Its syntax consists of closed (process) terms p, q that are constructed from a constant $\mathbf{0}$, a binary operator $_+_$ called *alternative composition*, and unary *prefix* operators $a_$, where a ranges over some nonempty set A of *actions.* Open terms t, u, v, w can moreover contain variables from a countably infinite set V (with typical elements x, y, z).

Transition rules: Intuitively, closed BCCSP(A) terms represent finite process behaviors, where $\mathbf{0}$ does not exhibit any behavior, $p + q$ is the nondeterministic choice between the behaviors of p and q, and ap executes action a to transform into p. This intuition is captured, in the style of Plotkin, by the transition rules below, which give rise to A-labeled transitions between closed terms.

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

Completed simulation preorder $\precsim_{\mathrm{CS}}$ and ready simulation preorder $\precsim_{\mathrm{RS}}$ constitute a *precongruence* for closed BCCSP(A)-terms. That is, $p_1 \precsim_{\mathrm{N}} q_1$ and $p_2 \precsim_{\mathrm{N}} q_2$ implies $ap_1 \precsim_{\mathrm{N}} aq_1$ for $a \in A$ and $p_1 + p_2 \precsim_{\mathrm{N}} q_1 + q_2$, where N ranges over $\{\mathrm{CS}, \mathrm{RS}\}$.

We extend the operational interpretation above to open terms by assuming that variables do not exhibit any behavior. For open terms t and u, we define $t \precsim_{\mathrm{N}} u$ (or $t \simeq_{\mathrm{N}} u$) if for any closed substitution σ, $\sigma(t) \precsim_{\mathrm{N}} \sigma(u)$ (or $\sigma(t) \simeq_{\mathrm{N}} \sigma(u)$, respectively).

Equations and inequations: Let axiomatization E be a collection of either inequations $t \preccurlyeq u$ or equations $t \approx u$. We write $E \vdash t \preccurlyeq u$ or $E \vdash t \approx u$ if this (in)equation can be derived from the (in)equations in E using the standard rules of (in)equational logic, where the rule for symmetry can be applied for equational derivations but not for inequational ones. A collection E of (in)equations is *sound* modulo a preorder $\precsim$ or equivalence $\simeq$ on closed terms if $(E \vdash p \preccurlyeq q) \Rightarrow p \precsim q$ or $(E \vdash p \approx q) \Rightarrow p \simeq q$, respectively, for all closed terms p and q. Vice versa, E is *ground-complete* modulo $\precsim$ or $\simeq$ if $p \precsim q \Rightarrow (E \vdash p \preccurlyeq q)$ or $p \simeq q \Rightarrow (E \vdash p \approx q)$, respectively, for all closed terms p and q. Finally, E is *ω-complete* modulo $\precsim$ or $\simeq$ if $t \precsim u \Rightarrow (E \vdash t \preccurlyeq u)$ or $t \simeq u \Rightarrow (E \vdash t \approx u)$, respectively for all open terms t and u.

The core axioms A1-4 [9] for BCCSP(A) below are ω-complete, and sound modulo bisimulation equivalence, which is the finest semantics in van Glabbeek's linear time - branching time spectrum (see Fig. 1).

$$\begin{array}{ll} \mathrm{A1} & x + y \approx y + x \\ \mathrm{A2} & (x + y) + z \approx x + (y + z) \\ \mathrm{A3} & x + x \approx x \\ \mathrm{A4} & x + \mathbf{0} \approx x \end{array}$$

In the remainder of this paper, process terms are considered modulo A1-2 and A4. A term x or at is a *summand* of each term $x + u$ or $at + u$, respectively. We use *summation* $\sum_{i \in \{i_1, \ldots, i_k\}} t_i$ (with $k \geq 0$) to denote $t_{i_1} + \cdots + t_{i_k}$, where the empty sum denotes $\mathbf{0}$.

As binding convention, alternative composition and summation bind weaker than prefixing. A (closed) substitution maps variables in V to (closed) terms. For every term t and substitution σ, the term $\sigma(t)$ is obtained by replacing every occurrence of a variable x in t by $\sigma(x)$.

3 Completed Similarity

In [6], van Glabbeek gave a finite equational axiomatization that is sound and ground-complete for BCCSP(A) modulo $\simeq_{\mathrm{CS}}$. It consists of axioms A1-4 together with

$$\mathrm{CS} \qquad a(bx+y+z) \approx a(bx+y+z) + a(bx+z)$$

where a, b range over A. Likewise, a finite sound and ground-complete axiomatization for BCCSP(A) modulo $\precsim_{\mathrm{CS}}$ is obtained by adding $bx+z \precsim_{\mathrm{CS}} bx+y+z$ to A1-4.

In this section we present a proof that the (in)equational theory of BCCSP(A) modulo completed similarity does not have a finite basis.

3.1 Completed Simulation Preorder

We start with proving that the inequational theory of BCCSP(A) modulo $\precsim_{\mathrm{CS}}$ does not have a finite basis. The corner stone for this negative result is the infinite family of inequations

$$a^n x \preccurlyeq a^n \mathbf{0} + a^n(x+y)$$

for $n \geq 1$. Here $a^n t$ denotes n prefixes of a: $a^0 t = t$ and $a^{n+1} t = a(a^n t)$. It is not hard to see that these inequations are sound modulo $\precsim_{\mathrm{CS}}$. The idea is that either x cannot perform any action, in which case $a^n x$ is completed simulated by $a^n\mathbf{0}$, or x can perform some action, in which case $a^n x$ is completed simulated by $a^n(x+y)$.

The *depth* of a term t, denoted by $depth(t)$, is the maximal number of transitions in sequence that t can exhibit. It is defined by: $depth(\mathbf{0}) = 0$, $depth(x) = 0$, $depth(t+u) = \max\{depth(t), depth(u)\}$, and $depth(at) = depth(t) + 1$.

Proposition 1. *Let E be a finite collection of inequations over* BCCSP(A) *that is sound modulo $\precsim_{\mathrm{CS}}$. Let n be larger than the depth of any term in E. Then from E we cannot derive the inequation*

$$a^n x \preccurlyeq a^n \mathbf{0} + a^n(x+y).$$

The main part of this section is devoted to proving Proposition 1. We start with two basic lemmas.

Let $t \overset{a_1 \cdots a_k}{\longrightarrow} t'$ (with $k \geq 0$) denote that there is a trace $t = t_0 \overset{a_1}{\to} t_1 \overset{a_2}{\to} \cdots \overset{a_k}{\to} t_k = t'$. If moreover $t' = x + t''$, then we say that x *occurs at depth* k in t. If t' cannot perform any transitions (meaning that each summand of t' is a variable or $\mathbf{0}$), then $t \overset{a_1 \cdots a_k}{\longrightarrow} t'$ is called a *termination trace* of t.

Lemma 1. *Let $t \precsim_{\mathrm{CS}} u$. If $t \overset{a_1 \cdots a_k}{\longrightarrow} x + t'$, then $u \overset{a_1 \cdots a_k}{\longrightarrow} x + u'$.*

Proof. Let $d > depth(u)$ and ρ a closed substitution such that $\rho(x) = a^d\mathbf{0}$ and $\rho(y) = \mathbf{0}$ for any variable $y \neq x$. By assumption, $t \overset{a_1 \cdots a_k}{\longrightarrow} x + t'$, so $\rho(t) \overset{a_1 \cdots a_{k+d}}{\longrightarrow} \mathbf{0}$

(with $a_{k+1}\cdots a_{k+d} = a^d$). Since $\rho(t) \precsim_{\rm CS} \rho(u)$, it follows that $\rho(u) \stackrel{a_1\cdots a_{k+d}}{\longrightarrow} v$ with $\mathbf{0} \precsim_{\rm CS} v$, which implies $v \simeq_{\rm CS} \mathbf{0}$. Since $d > \mathit{depth}(u)$, clearly $u \stackrel{a_1\cdots a_i}{\longrightarrow} y + u'$ where $\rho(y) \stackrel{a_{i+1}\cdots a_{k+d}}{\longrightarrow} v$. We have $i \leq \mathit{depth}(u) < d$, so $\rho(y) \neq \mathbf{0}$, and hence $y = x$ and $i = k$. Concluding, $u \stackrel{a_1\cdots a_k}{\longrightarrow} x + u'$. □

Lemma 2. *If $at \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$, then at is completed similar to $a^n\mathbf{0}$, $a^n x$, $a^n y$ or $a^n(x+y)$.*

Proof. By assumption, $at \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$. Then clearly every termination trace of t has length $n-1$, and executes only a's. Moreover, by Lemma 1, t can only contain the variables x and y. It follows that for every trace of t such that $t \stackrel{a^{n-1}}{\longrightarrow} t'$, t' is completed similar to either $\mathbf{0}, x, y$ or $x+y$. Suppose, towards a contradiction, that $t \stackrel{a^{n-1}}{\longrightarrow} t_1$ and $t \stackrel{a^{n-1}}{\longrightarrow} t_2$ with $t_1 \not\simeq_{\rm CS} t_2$. In each of the six possible cases (modulo symmetry) we give a closed substitution ρ with $\rho(at) \not\precsim_{\rm CS} \rho(a^n\mathbf{0} + a^n(x+y))$.

- Cases 1,2,3: $t_1 \simeq_{\rm CS} \mathbf{0}$ and $t_2 \simeq_{\rm CS} x$, y or $x+y$. Let $\rho(x) \not\simeq_{\rm CS} \mathbf{0}$ and $\rho(y) \not\simeq_{\rm CS} \mathbf{0}$. Then $\rho(t) \not\precsim_{\rm CS} a^{n-1}\mathbf{0}$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_2) \not\simeq_{\rm CS} \mathbf{0}$) and $\rho(t) \not\precsim_{\rm CS} a^{n-1}\rho(x+y)$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_1) \simeq_{\rm CS} \mathbf{0}$ and $\rho(x+y) \not\simeq_{\rm CS} \mathbf{0}$).
- Cases 4,5: $t_1 \simeq_{\rm CS} x$ and $t_2 \simeq_{\rm CS} y$ or $x+y$. Let $\rho(x) = \mathbf{0}$ and $\rho(y) \not\simeq_{\rm CS} \mathbf{0}$. Then $\rho(t) \not\precsim_{\rm CS} a^{n-1}\mathbf{0}$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_2) \not\simeq_{\rm CS} \mathbf{0}$) and $\rho(t) \not\precsim_{\rm CS} a^{n-1}\rho(x+y)$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_1) \simeq_{\rm CS} \mathbf{0}$ and $\rho(x+y) \not\simeq_{\rm CS} \mathbf{0}$).
- Case 6: $t_1 \simeq_{\rm CS} y$ and $t_2 \simeq_{\rm CS} x+y$. Let $\rho(x) \not\simeq_{\rm CS} \mathbf{0}$ and $\rho(y) = \mathbf{0}$. Then $\rho(t) \not\precsim_{\rm CS} a^{n-1}\mathbf{0}$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_2) \not\simeq_{\rm CS} \mathbf{0}$) and $\rho(t) \not\precsim_{\rm CS} a^{n-1}\rho(x+y)$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_1) \simeq_{\rm CS} \mathbf{0}$ and $\rho(x+y) \not\simeq_{\rm CS} \mathbf{0}$).

We conclude that the six cases above all contradict $at \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$. Hence it must be the case that for each pair of traces $t \stackrel{a^{n-1}}{\longrightarrow} t_1$ and $t \stackrel{a^{n-1}}{\longrightarrow} t_2$, $t_1 \simeq_{\rm CS} t_2$. Moreover, by Lemma 1, t does not contain variables at depths smaller than $n-1$. It is not hard to see that this implies the lemma. □

The following key lemma paves the way for the proof of Proposition 1.

Lemma 3. *Let E be a finite collection of inequations over* BCCSP(A) *that is sound modulo $\precsim_{\rm CS}$. Let n be greater than the depth of any term in E. Assume that:*

- *$E \vdash t \preccurlyeq u$;*
- *$u \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$; and*
- *t has a summand completed similar to $a^n x$.*

Then u has a summand completed similar to $a^n x$.

Proof. By induction on the depth of the proof of the inequation $t \preccurlyeq u$ from E. We proceed by a case analysis on the last rule used in the proof of $t \preccurlyeq u$ from E.

- CASE 1: $E \vdash t \preccurlyeq u$ because $\sigma(v) = t$ and $\sigma(w) = u$ for some $v \preccurlyeq w \in E$ and substitution σ.
  Since $t = \sigma(v)$ has a summand completed similar to $a^n x$, we can distinguish two cases.
  - CASE 1.1: v has as summand some variable z where $\sigma(z)$ has a summand completed similar to $a^n x$.
    Since v has z as summand, and soundness of E yields $v \precsim_{\rm CS} w$, by Lemma 1, w also has z as summand. Then clearly $u = \sigma(w)$ has a summand completed similar to $a^n x$.
  - CASE 1.2: v has a summand av' where $\sigma(av') \simeq_{\rm CS} a^n x$.
    Since n is larger than the depth of v, *depth*$(av') < n$. So, since $\sigma(av') \simeq_{\rm CS} a^n x$, $av' \stackrel{a^k}{\to} z + v''$ where $1 \leq k < n$ and $\sigma(z) \simeq_{\rm CS} a^{n-k} x$. Since $v \precsim_{\rm CS} w$, by Lemma 1, w has a summand aw' such that $w' \stackrel{a^{k-1}}{\to} z + w''$, and consequently $\sigma(w') \stackrel{a^{n-1}}{\to} w'''$ with $w''' \simeq_{\rm CS} x$. Furthermore, $a\sigma(w') \precsim_{\rm CS} \sigma(w) \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$. Then Lemma 2 yields $\sigma(w') \simeq_{\rm CS} a^{n-1}x$. Hence $\sigma(aw') \simeq_{\rm CS} a^n x$. So $u = \sigma(w)$ has a summand completed similar to $a^n x$.

- CASE 2: $E \vdash t \preccurlyeq u$ by reflexivity. Then $t = u$, so u trivially has a summand completed similar to $a^n x$.

- CASE 3: $E \vdash t \preccurlyeq u$ by transitivity.
  Then $E \vdash t \preccurlyeq v$ and $E \vdash v \preccurlyeq u$ for some term v. By the soundness of E, $v \precsim_{\rm CS} u \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$. So by induction, v has a summand completed similar to $a^n x$. Hence, again by induction, u has a summand completed similar to $a^n x$.

- CASE 4: $E \vdash t \preccurlyeq u$ because $t = t' + t''$ and $u = u' + u''$ for some t', u', t'', u'' such that $E \vdash t' \preccurlyeq u'$ and $E \vdash t'' \preccurlyeq u''$.
  Since t has a summand completed similar to $a^n x$, so does either t' or t''. Assume, without loss of generality, that t' has a summand completed similar to $a^n x$. Then clearly $u' \not\precsim_{\rm CS} \mathbf{0}$. So, since $u \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$, it follows that $u' \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$. By induction, u' (and thus u) has a summand completed similar to $a^n x$.

- CASE 5: $E \vdash t \preccurlyeq u$ because $t = at'$ and $u = au'$ for some t', u' such that $E \vdash t' \preccurlyeq u'$.
  Since $t = at'$ consists of a single summand, $at' \simeq_{\rm CS} a^n x$. By the soundness of E, $a^n x \precsim_{\rm CS} au'$. Since moreover $au' \precsim_{\rm CS} a^n\mathbf{0} + a^n(x+y)$, Lemma 2 yields $u = au' \simeq_{\rm CS} a^n x$. □

Now we are in a position to prove **Proposition 1**.

Proof. Let E be a finite collection of inequations over BCCSP(A) that is sound modulo $\precsim_{\rm CS}$. Let n be larger than the depth of any term in E.

$a^n\mathbf{0} + a^n(x + y)$ does not contain a summand completed similar to $a^n x$. So according to Lemma 3, the inequation $a^n x \preccurlyeq a^n\mathbf{0} + a^n(x + y)$, which is sound modulo $\precsim_{\mathrm{CS}}$, cannot be derived from E. □

Theorem 1. $\precsim_{\mathrm{CS}}$ *is not finitely based over* BCCSP(A).

Proof. By Proposition 1, no finite collection of inequations over BCCSP(A) that is sound modulo $\precsim_{\mathrm{CS}}$ proves all inequations that are sound modulo $\precsim_{\mathrm{CS}}$. □

3.2 Completed Simulation Equivalence

Following the same line as in Section 3.1, we can prove that the equational theory of BCCSP(A) modulo $\simeq_{\mathrm{CS}}$ does not have a finite basis. The proofs are similar to the proofs of the corresponding results in the previous section.

Lemma 4. *Let E be a finite collection of equations over* BCCSP(A) *that is sound modulo $\simeq_{\mathrm{CS}}$. Let n be greater than the depth of any term in E. Assume that:*

- *$E \vdash t \approx u$;*
- *$u \simeq_{\mathrm{CS}} a^n\mathbf{0} + a^n(x + y)$; and*
- *t has a summand completed similar to $a^n x$.*

Then u has a summand completed similar to $a^n x$.

Proof. By induction on the depth of the proof of the equation $t \approx u$ from E. First note that by postulating that for each axiom in E also its symmetric counterpart is present in E, one may assume that, without loss of generality, applications of symmetry happen first in equational proof. Thus in the proof, we can tacitly assume that equational axiomatization E is closed with respect to symmetry.

Now the proof proceeds by a case analysis on the last rule used in the proof of $t \approx u$ from E, similar to the proof of Lemma 3. This case analysis is omitted here. □

Proposition 2. *Let E be a finite collection of equations over* BCCSP(A) *that is sound modulo $\simeq_{\mathrm{CS}}$. Let n be larger than the depth of any term in E. Then from E we cannot derive the equation*

$$a^n x + a^n\mathbf{0} + a^n(x + y) \approx a^n\mathbf{0} + a^n(x + y)$$

Proof. $a^n\mathbf{0} + a^n(x + y)$ does not contain a summand completed similar to $a^n x$. So according to Lemma 4, the equation $a^n x + a^n\mathbf{0} + a^n(x+y) \approx a^n\mathbf{0} + a^n(x+y)$, which is sound modulo $\simeq_{\mathrm{CS}}$, cannot be derived from E. □

Theorem 2. $\simeq_{\mathrm{CS}}$ *is not finitely based over* BCCSP(A).

Proof. By Proposition 2, no finite collection of equations over BCCSP(A) that is sound modulo $\simeq_{\mathrm{CS}}$ proves all equations that are sound modulo $\simeq_{\mathrm{CS}}$. □

4 Ready Similarity

Blom, Fokkink and Nain [2] gave a finite equational axiomatization that is sound and ground-complete for BCCSP(A) modulo $\simeq_{\mathrm{RS}}$. It consists of axioms A1-4 together with

$$\text{RS} \qquad a(bx + by + z) \approx a(bx + by + z) + a(bx + z)$$

where a, b range over A. If A is infinite, then Groote's technique of inverted substitutions from [7] can be applied in a straightforward fashion to prove that this axiomatization is ω-complete. So in this case, ready simulation equivalence is finitely based over BCCSP(A). This finite basis can be adapted in a straightforward fashion to a finite basis for BCCSP(A) modulo ready simulation preorder (simply add $bx + z \precsim_{\mathrm{RS}} bx + by + z$ to A1-4).

In this section we prove that if A is finite, then ready simulation preorder and equivalence are not finitely based over BCCSP(A). The infinite family of equations, and the structure of the proof, are very similar to the case of completed similarity in the previous section (where we obtained a negative result for arbitrary alphabets).

4.1 Ready Simulation Preorder with $|A| < \infty$

First we present a proof that if A is finite, then the inequational theory of BCCSP(A) modulo $\precsim_{\mathrm{RS}}$ does not have a finite basis. The corner stone for this negative result is the infinite family of inequations

$$a^n x \preccurlyeq a^n \mathbf{0} + \sum_{b \in A} a^n (x + b\mathbf{0})$$

for $n \geq 1$. It is not hard to see that these inequations are sound modulo $\precsim_{\mathrm{RS}}$. The idea is that either x cannot perform any action, in which case $a^n x$ is ready simulated by $a^n \mathbf{0}$, or x can perform some action b, in which case $a^n x$ is ready simulated by $a^n (x + b\mathbf{0})$.

Proposition 3. *Let $|A| < \infty$. Let E be a finite collection of inequations over* BCCSP(A) *that is sound modulo $\precsim_{\mathrm{RS}}$. Let n be larger than the depth of any term in E. Then from E we cannot derive the inequation*

$$a^n x \preccurlyeq a^n \mathbf{0} + \sum_{b \in A} a^n (x + b\mathbf{0}).$$

The main part of this section is devoted to proving Proposition 3. Note that Lemma 1 also applies to ready simulation preorder, as it is finer than completed simulation preorder.

Lemma 5. *Let $|A| < \infty$. If $at \precsim_{\mathrm{RS}} a^n \mathbf{0} + \sum_{b \in A} a^n (x + b\mathbf{0})$, then at is ready similar to $a^n \mathbf{0}$, $a^n x$ or $a^n (x + b\mathbf{0})$ for some $b \in A$.*

Proof. By assumption, $at \precsim_{\rm RS} a^n\mathbf{0} + \sum_{b\in A} a^n(x+b\mathbf{0})$. Then clearly every termination trace of t is of the form $t \stackrel{a^{n-1}}{\longrightarrow} t'$ or $t \stackrel{a^{n-1}b}{\longrightarrow} t'$. Moreover, by Lemma 1, t can only contain the variable x, and x cannot occur at depth n in t. It follows that for every trace of t such that $t \stackrel{a^{n-1}}{\longrightarrow} t'$, t' is ready similar to either $\mathbf{0}$, x or $x+b_0\mathbf{0}$ for some $b_0 \in A$. Suppose, towards a contradiction, that $t \stackrel{a^{n-1}}{\longrightarrow} t_1$ and $t \stackrel{a^{n-1}}{\longrightarrow} t_2$ with $t_1 \not\simeq_{\rm RS} t_2$. In each of the four possible cases (modulo symmetry) we give a closed substitution ρ with $\rho(at) \not\precsim_{\rm RS} \rho(a^n\mathbf{0} + \sum_{b\in A} a^n(x+b\mathbf{0}))$.

- Cases 1,2: $t_1 \simeq_{\rm RS} \mathbf{0}$ and $t_2 \simeq_{\rm RS} x$ or $x+b_0\mathbf{0}$ for some $b_0 \in A$. Let $\rho(x) \not\simeq_{\rm RS} \mathbf{0}$. Then $\rho(t) \not\precsim_{\rm RS} a^{n-1}\mathbf{0}$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_2) \not\simeq_{\rm RS} \mathbf{0}$) and $\rho(t) \not\precsim_{\rm RS} a^{n-1}\rho(x+b\mathbf{0})$ for each $b\in A$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_1) \simeq_{\rm RS} \mathbf{0}$ and $\rho(x+b\mathbf{0}) \not\simeq_{\rm RS} \mathbf{0}$).
- Case 3: $t_1 \simeq_{\rm RS} x$ and $t_2 \simeq_{\rm RS} x+b_0\mathbf{0}$ for some $b_0\in A$. Let $\rho(x)=\mathbf{0}$. Then $\rho(t) \not\precsim_{\rm RS} a^{n-1}\mathbf{0}$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_2) \not\simeq_{\rm RS} \mathbf{0}$) and $\rho(t) \not\precsim_{\rm RS} a^{n-1}\rho(x+b\mathbf{0})$ for each $b\in A$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_1) \simeq_{\rm RS} \mathbf{0}$ and $\rho(x+b\mathbf{0}) \not\simeq_{\rm RS} \mathbf{0}$).
- Case 4: $t_1 \simeq_{\rm RS} x+b_0\mathbf{0}$ and $t_2 \simeq_{\rm RS} x+b_1\mathbf{0}$ for some $b_0,b_1\in A$ with $b_0\neq b_1$. Let $\rho(x)=\mathbf{0}$. Then $\rho(t) \not\precsim_{\rm RS} a^{n-1}\mathbf{0}$ (because $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_1) \not\simeq_{\rm RS} \mathbf{0}$) and $\rho(t) \not\precsim_{\rm RS} a^{n-1}\rho(x+b\mathbf{0})$ for each $b\in A$ (because $b\neq b_i$ for $i=0$ or $i=1$, so that $\rho(t) \stackrel{a^{n-1}}{\longrightarrow} \rho(t_i) \simeq_{\rm RS} b_i\mathbf{0}$ and $\rho(x+b\mathbf{0}) \not\simeq_{\rm RS} b_i\mathbf{0}$).

We conclude that the four cases above all contradict $at \precsim_{\rm RS} a^n\mathbf{0} + \sum_{b\in A} a^n(x+b\mathbf{0})$. Hence it must be the case that for each pair of traces $t \stackrel{a^{n-1}}{\longrightarrow} t_1$ and $t \stackrel{a^{n-1}}{\longrightarrow} t_2$, $t_1 \simeq_{\rm RS} t_2$. Moreover, by Lemma 1, t does not contain variables at depths smaller than $n-1$. It is not hard to see that this implies the lemma. □

The following key lemma paves the way for the proof of Proposition 3.

Lemma 6. *Let $|A|<\infty$. Let E be a finite collection of inequations over* BCCSP (A) *that is sound modulo $\precsim_{\rm RS}$. Let n be greater than the depth of any term in E. Assume that:*

- *$E \vdash t \preccurlyeq u$;*
- *$u \precsim_{\rm RS} a^n\mathbf{0} + \sum_{b\in A} a^n(x+b\mathbf{0})$; and*
- *t has a summand ready similar to $a^n x$.*

Then u has a summand ready similar to $a^n x$.

Proof. By induction on the depth of the proof of the inequation $t \preccurlyeq u$ from E. We proceed by a case analysis on the last rule used in the proof of $t \preccurlyeq u$ from E.

- Case 1: $E \vdash t \preccurlyeq u$ because $\sigma(v)=t$ and $\sigma(w)=u$ for some $v \preccurlyeq w \in E$ and substitution σ.
  Since $t=\sigma(v)$ has a summand ready similar to $a^n x$, we can distinguish two cases.

- CASE 1.1: v has as summand some variable z where $\sigma(z)$ has a summand ready similar to $a^n x$.
  Since v has z as summand, and soundness of E yields $v \precsim_{\mathrm{RS}} w$, by Lemma 1, w also has z as summand. Then clearly $u = \sigma(w)$ has a summand ready similar to $a^n x$.
- CASE 1.2: v has a summand av' where $\sigma(av') \simeq_{\mathrm{RS}} a^n x$.
  Since n is larger than the depth of v, $depth(av') < n$. So, since $\sigma(av') \simeq_{\mathrm{RS}} a^n x$, $av' \stackrel{a^k}{\to} z + v''$ where $1 \leq k < n$ and $\sigma(z) \simeq_{\mathrm{RS}} a^{n-k}x$. Since $v \precsim_{\mathrm{RS}} w$, by Lemma 1, w has a summand aw' such that $w' \stackrel{a^{k-1}}{\to} z + w''$, and consequently $\sigma(w') \stackrel{a^{n-1}}{\to} w'''$ with $w''' \simeq_{\mathrm{RS}} x$. Furthermore, $a\sigma(w') \precsim_{\mathrm{RS}} \sigma(w) \precsim_{\mathrm{RS}} a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$. Then Lemma 5 yields $\sigma(w') \simeq_{\mathrm{RS}} a^{n-1}x$. Hence $\sigma(aw') \simeq_{\mathrm{RS}} a^n x$. So $u = \sigma(w)$ has a summand ready similar to $a^n x$.

– CASE 2: $E \vdash t \preccurlyeq u$ by reflexivity. Then $t = u$, so u trivially has a summand ready similar to $a^n x$.

– CASE 3: $E \vdash t \preccurlyeq u$ by transitivity.
  Then $E \vdash t \preccurlyeq v$ and $E \vdash v \preccurlyeq u$ for some term v. By the soundness of E, $v \precsim_{\mathrm{RS}} u \precsim_{\mathrm{RS}} a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$. So by induction, v has a summand ready similar to $a^n x$. Hence, again by induction, u has a summand ready similar to $a^n x$.

– CASE 4: $E \vdash t \preccurlyeq u$ because $t = t' + t''$ and $u = u' + u''$ for some t', u', t'', u'' such that $E \vdash t' \preccurlyeq u'$ and $E \vdash t'' \preccurlyeq u''$.
  Since t has a summand ready similar to $a^n x$, so does either t' or t''. Assume, without loss of generality, that t' has a summand ready similar to $a^n x$. Then clearly $u' \not\precsim_{\mathrm{RS}} \mathbf{0}$. So, since $u \precsim_{\mathrm{RS}} a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$, it follows that $u' \precsim_{\mathrm{RS}} a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$. By induction, u' (and thus u) has a summand ready similar to $a^n x$.

– CASE 5: $E \vdash t \preccurlyeq u$ because $t = at'$ and $u = au'$ for some t', u' such that $E \vdash t' \preccurlyeq u'$.
  Since $t = at'$ consists of a single summand, $at' \simeq_{\mathrm{RS}} a^n x$. By the soundness of E, $a^n x \precsim_{\mathrm{RS}} au'$. Since moreover $au' \precsim_{\mathrm{RS}} a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$, Lemma 5 yields $u = au' \simeq_{\mathrm{RS}} a^n x$. □

Now we are in a position to prove **Proposition 3**.

Proof. Let E be a finite collection of inequations over BCCSP(A) that is sound modulo $\precsim_{\mathrm{RS}}$. Let n be larger than the depth of any term in E.

$a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$ does not contain a summand ready similar to $a^n x$. So according to Lemma 6, the inequation $a^n x \preccurlyeq a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$, which is sound modulo $\precsim_{\mathrm{RS}}$, cannot be derived from E. □

Theorem 3. *Let* $|A| < \infty$. *Then* $\precsim_{\mathrm{RS}}$ *is not finitely based over* BCCSP(A).

Proof. By Proposition 3, no finite collection of inequations over BCCSP(A) that is sound modulo $\precsim_{\mathrm{RS}}$ proves all inequations that are sound modulo $\precsim_{\mathrm{RS}}$. □

4.2 Ready Simulation Equivalence with $|A| < \infty$

Following the same line as in Section 4.1, we can prove that if A is finite, then the equational theory of BCCSP(A) modulo $\simeq_{\mathrm{RS}}$ does not have a finite basis. The proofs are similar to the proofs of the corresponding results in the previous section.

Lemma 7. *Let $|A| < \infty$. Let E be a finite collection of equations over* BCCSP (A) *that is sound modulo $\simeq_{\mathrm{RS}}$. Let n be greater than the depth of any term in E. Assume that:*

- *$E \vdash t \approx u$;*
- *$u \simeq_{\mathrm{RS}} a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$; and*
- *t has a summand ready similar to $a^n x$.*

Then u has a summand ready similar to $a^n x$.

Proof. By induction on the depth of the proof of the equation $t \approx u$ from E. Recall that as in the proof of Lemma 4, without loss of generality, we may assume that applications of symmetry happen first in equational proof, i.e. E is closed with respect to symmetry.

Now the proof proceeds by a case analysis on the last rule used in the proof of $t \approx u$ from E, similar to the proof of Lemma 6. This case analysis is omitted here. □

Proposition 4. *Let $|A| < \infty$. Let E be a finite collection of equations over* BCCSP(A) *that is sound modulo $\simeq_{\mathrm{RS}}$. Let n be larger than the depth of any term in E. Then from E we cannot derive the equation*

$$a^n x + a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0}) \approx a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$$

Proof. Let E be a finite collection of equations over BCCSP(A) that is sound modulo $\simeq_{\mathrm{RS}}$. Let n be larger than the depth of any term in E.

$a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$ does not contain a summand ready similar to $a^n x$. So according to Lemma 7, the equation $a^n x + a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0}) \approx a^n\mathbf{0} + \sum_{b\in A} a^n(x + b\mathbf{0})$, which is sound modulo $\simeq_{\mathrm{RS}}$, cannot be derived from E. □

Theorem 4. *Let $|A| < \infty$. Then $\simeq_{\mathrm{RS}}$ is not finitely based over* BCCSP(A).

Proof. By Proposition 4, no finite collection of equations over BCCSP(A) that is sound modulo $\simeq_{\mathrm{RS}}$ proves all equations that are sound modulo $\simeq_{\mathrm{RS}}$. □

4.3 Ready Simulation Equivalence with $|A| = \infty$

In this section we prove that if A is infinite, then the axiomatization A1-4 together with

$$\mathrm{RS} \qquad a(bx + by + z) \approx a(bx + by + z) + a(bx + z)$$

from [2], which is ground-complete for BCCSP(A) modulo $\simeq_{\mathrm{RS}}$, is ω-complete. The proof is based on inverted substitutions; this technique, which is due to Groote [7], works as follows. Consider an axiomatization E. For each equation $t \approx u$ of which all closed instances can be derived from E, one must define a closed substitution ρ and a mapping $R : \mathrm{T(BCCSP)} \to \mathbb{T}(\mathrm{BCCSP})$ such that:

(1) $E \vdash R(\rho(t)) \approx t$ and $E \vdash R(\rho(u)) \approx u$;

(2) for each function symbol f (with arity n), $E \cup \{p_i \approx q_i,\ R(p_i) \approx R(q_i) \mid i = 1, \ldots, n\} \vdash R(f(p_1, \ldots, p_n)) \approx R(f(q_1, \ldots, q_n))$ for all closed terms $p_1, \ldots, p_n, q_1, \ldots, q_n$; and

(3) $E \vdash R(\sigma(v)) \approx R(\sigma(w))$ for each $v \approx w \in E$ and closed substitution σ.

Then, as proved in [7], E is ω-complete.

Theorem 5. *If $|A| = \infty$, then A1-4+RS is ω-complete.*

Proof. Consider two terms $t, u \in \mathbb{T}(\mathrm{BCCSP})$. Define $\rho : V \to \mathrm{T(BCCSP)}$ by $\rho(x) = a_x\mathbf{0}$, where a_x is a unique action for $x \in V$ that occurs in neither t nor u. Such actions exist because A is infinite. We define $R : \mathrm{T(BCCSP)} \to \mathbb{T}(\mathrm{BCCSP})$ as follows:

$$\begin{cases} R(\mathbf{0}) &= \mathbf{0} \\ R(at) &= aR(t) \text{ if } a \neq a_x \text{ for all } x \in V \\ R(a_x t) &= x \\ R(t+u) &= R(t) + R(u) \end{cases}$$

We now check the three properties from [7]:

(1) Since t and u do not contain actions of the form a_x, clearly $R(\rho(t)) = t$.

(2) Consider the operator $_ + _$. From $R(p_1) \approx R(q_1)$ and $R(p_2) \approx R(q_2)$ we derive $R(p_1 + p_2) = R(p_1) + R(p_2) \approx R(q_1) + R(q_2) = R(q_1 + q_2)$.

Consider the prefix operator $a_$. We distinguish two cases.
- $a \neq a_y$ for all $y \in V$. Then from $R(p_1) \approx R(q_1)$ we derive $R(ap_1) = aR(p_1) \approx aR(q_1) = R(aq_1)$.
- $a = a_y$ for some $y \in V$. Then $R(a_y p_1) = y = R(a_y q_1)$.

(3) For A1-4, the proof is trivial. We check the remaining case RS. Let σ be a closed substitution. We consider three cases.
- $a = a_y$ for some $y \in V$.
Then $R(a_y(b\sigma(x_1) + b\sigma(x_2) + \sigma(x_3))) = y \approx y + y = R(a_y(b\sigma(x_1) + b\sigma(x_2) + \sigma(x_3)) + a_y(b\sigma(x_1) + \sigma(x_3)))$.
- $a \neq a_y$ for all $y \in V$ and $b = b_z$ for some $z \in V$.
Then $R(a(b_z\sigma(x_1) + b_z\sigma(x_2) + \sigma(x_3))) = a(z + z + R(\sigma(x_3))) \approx a(z + z + R(\sigma(x_3))) + a(z + R(\sigma(x_3))) = R(a(b_z\sigma(x_1) + b_z\sigma(x_2) + \sigma(x_3)) + a(b_z\sigma(x_1) + \sigma(x_3)))$.
- $a \neq a_y$ for all $y \in V$ and $b \neq b_z$ for all $z \in V$.
Then $R(a(b\sigma(x_1)+b\sigma(x_2)+\sigma(x_3)))=a(bR(\sigma(x_1))+bR(\sigma(x_2))+R(\sigma(x_3))) \approx a(bR(\sigma(x_1)) + bR(\sigma(x_2)) + R(\sigma(x_3))) + a(bR(\sigma(x_1)) + R(\sigma(x_3))) = R(a(b\sigma(x_1) + b\sigma(x_2) + \sigma(x_3)) + a(b\sigma(x_1) + \sigma(x_3)))$. $\square$

References

1. L. Aceto, W.J. Fokkink, R.J. van Glabbeek, and A. Ingolfsdottir. Nested semantics over finite trees are equationally hard. *Information and Computation*, 191(2): 203–232, 2004.
2. S.C.C. Blom, W.J. Fokkink, and S. Nain. On the axiomatizability of ready traces, ready simulation and failure traces. In *Proceedings 30th Colloquium on Automata, Languages and Programming (ICALP'03)*, Eindhoven, LNCS 2719, pp. 109–118. Springer, 2003.
3. W.J. Fokkink and S.P. Luttik. An ω-complete equational specification of interleaving. In *Proceedings 27th Colloquium on Automata, Languages and Programming (ICALP'00)*, Geneva, LNCS 1853, pp. 729–743. Springer, 2000.
4. W.J. Fokkink and S. Nain. On finite alphabets and infinite bases: From ready pairs to possible worlds. In *Proceedings 7th Conference on Foundations of Software Science and Computation Structures (FOSSACS'04)*, Barcelona, LNCS 2987, pp. 182–194. Springer, 2004.
5. W.J. Fokkink and S. Nain. A finite basis for failure semantics. In *Proceedings 32nd Colloquium on Automata, Languages and Programming (ICALP'05)*, Lisbon, LNCS 3580, pp. 755–765. Springer, 2005.
6. R.J. van Glabbeek. The linear time – branching time spectrum I. The semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, eds, *Handbook of Process Algebra*, pp. 3–99. Elsevier, 2001.
7. J.F. Groote. A new strategy for proving ω-completeness with applications in process algebra. In *Proceedings 1st Conference on Concurrency Theory (CONCUR'90)*, Amsterdam, LNCS 458, pp. 314–331. Springer, 1990.
8. H. Lin. PAM: A process algebra manipulator. *Formal Methods in System Design*, 7(3):243–259, 1995.
9. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
10. F. Moller. *Axioms for Concurrency.* PhD thesis, University of Edinburgh, 1989.

A Theory for Observational Fault Tolerance
(Extended Abstract)

Adrian Francalanza[1] and Matthew Hennessy[2]

[1] University of Malta, Msida MSD 06, Malta
afra1@um.edu.mt
[2] University of Sussex, Brighton BN1 9RH, England
matthewh@sussex.ac.uk

Abstract. In general, faults cannot be prevented; instead, they need to be tolerated to guarantee certain degrees of software dependability. We develop a theory for fault tolerance for a distributed pi-calculus, whereby locations act as units of failure and redundancy is distributed across independently failing locations. We give formal definitions for fault tolerant programs in our calculus, based on the well studied notion of contextual equivalence. We then develop bisimulation proof techniques to verify fault tolerance properties of distributed programs and show they are sound with respect to our definitions for fault tolerance.

1 Introduction

One reason for the study of programs in the presence of *faults*, i.e. defects at the lowest level of abstractions [2], is to be able to construct more *dependable* systems, meaning systems exhibiting a high probability of *behaving* according to their *specification* [13]. System dependability is often expressed through attributes like maintainability, availability, safety and *reliability*, the latter of which is defined as a measure of the *continuous delivery* of *correct behaviour*, [13]. There are a number of approaches for achieving system dependability in the presence of faults, ranging from fault removal, fault prevention and *fault tolerance*.

The fault tolerant approach to system dependability consist of various techniques that employ *redundancy* to prevent faults from generating *failure*, i.e. abnormal behaviour caused by faults [2]. Two forms of redundancy are *space redundancy* (*replication*), i.e. using several copies of the same system components, and *time redundancy*, i.e. performing the same chunk of computation more than once. Certain fault tolerant techniques are based on *fault detection* which subsequently trigger *fault recovery*. If enough redundancy is used, fault recovery can lead to *fault masking*, where the specified behaviour is preserved without noticeable glitch.

Fault tolerance is of particular relevance in distributed computing; distribution yield a natural notion of *partial failure*, whereby faults affect a *subset* of the computation. Partial failure, in turn, gives scope for introducing redundancy as *replication*, distributed across independently failing entities such as locations. In general, the higher the replication, the greater the potential for fault tolerance. Nevertheless, fault tolerance also depends on how replicas are managed.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 16–31, 2006.
© Springer-Verlag Berlin Heidelberg 2006

One classification, due to [13], identifies three classes, namely *active replication* (all replicas are invoked for every operation), *passive replication* (operations are invoked on primary replicas and secondary replicas are updated in batches at checkpoints), and *lazy replication* (a hybrid of the previous two, exploiting the separation between write and read operations).

In this paper we address fault tolerance in a distributed setting, focussing on simple examples using *stateless* (read-only) replicas which are invoked only once. We code these examples in Dπ [8] with failing locations [5], a simple distributed version of the standard π-calculus [11], where the locations that host processes model closely physical network nodes.

Example 1. Consider the systems server_i, three server implementations accepting client requests on channel *req* with two arguments, x being the value to process and y being the reply channel on which the answer is returned.

$$\mathsf{server}_1 \Leftarrow (\nu\, data) \begin{pmatrix} l[\![req?(x,y).\mathsf{go}\; k_1.data!\langle x,y,l\rangle]\!] \\ \mid k_1[\![data?(x,y,z).\mathsf{go}\; z.y!\langle f(x)\rangle]\!] \end{pmatrix}$$

$$\mathsf{server}_2 \Leftarrow (\nu\, data) \begin{pmatrix} l \left[\!\!\left[req?(x,y).(\nu sync) \begin{pmatrix} \mathsf{go}\; k_1.data!\langle x, sync, l\rangle \\ \mid \mathsf{go}\; k_2.data!\langle x, sync, l\rangle \\ \mid sync?(x).y!\langle x\rangle \end{pmatrix} \right]\!\!\right] \\ \mid k_1[\![data?(x,y,z).\mathsf{go}\; z.y!\langle f(x)\rangle]\!] \\ \mid k_2[\![data?(x,y,z).\mathsf{go}\; z.y!\langle f(x)\rangle]\!] \end{pmatrix}$$

$$\mathsf{server}_3 \Leftarrow (\nu\, data) \begin{pmatrix} l \left[\!\!\left[req?(x,y).(\nu sync) \begin{pmatrix} \mathsf{go}\; k_1.data!\langle x, sync, l\rangle \\ \mid \mathsf{go}\; k_2.data!\langle x, sync, l\rangle \\ \mid \mathsf{go}\; k_3.data!\langle x, sync, l\rangle \\ \mid sync?(x).y!\langle x\rangle \end{pmatrix} \right]\!\!\right] \\ \mid k_1[\![data?(x,y,z).\mathsf{go}\; z.y!\langle f(x)\rangle]\!] \\ \mid k_2[\![data?(x,y,z).\mathsf{go}\; z.y!\langle f(x)\rangle]\!] \\ \mid k_3[\![data?(x,y,z).\mathsf{go}\; z.y!\langle f(x)\rangle]\!] \end{pmatrix}$$

Requests are forwarded to internal databases, denoted by the scoped channel *data*, distributed and replicated across the auxiliary locations k_i. A database looks up the mapping of the value x using some unspecified function $f(-)$ and returns the answer, $f(x)$, back on port y. When multiple replicas are used, as in $\mathsf{server}_{2,3}$, requests are sent to all replicas in an arbitrary fashion, without the use of failure detection, and multiple answers are synchronised at l on the scoped channel *sync*, returning the first answer received on y.

The theory developed in [5] enables us to *differentiate* between these systems, based on the different behaviour observed when composed with systems such as

$$\mathsf{client} \Leftarrow l[\![req!\langle v, ret\rangle]\!]$$

in a setting where locations may fail. Here we go one step further, allowing us to *quantify* in some sense the difference between these systems. Intuitively,

if locations $k_i, i = 1, 2, 3$, can fail in fail-stop fashion[12] and observations are limited to location l only, then server_2 seems to be more *fault tolerant* than server_1; observers limited to l, such as client, cannot observe changes in behaviour in server_2 when *at most 1* location from k_i fails. Similarly, server_3 is more *fault tolerant* than server_1 and server_2 because $\mathsf{server}_3 \,|\, \mathsf{client}$ preserves its behaviour at l *up to 2* faults occurring at $k_{1..3}$.

In this paper we give a formal definition of when a system is deemed to be fault tolerant up to n-faults, which coincides with this intuition. As in [5] we need to consider systems M, running on some network, which we will represent as $\Gamma \triangleright M$. Then we will say that M is fault-tolerant up to n faults if

$$F^n[\Gamma \triangleright M] \cong \Gamma \triangleright M \quad (1)$$

where $F^n[\ ]$ is some context which induces at most n faults, and $\cong$ is some behavioural equivalence between systems descriptions. A key aspect of this behavioural equivalence is the implicit separation between *reliable* locations, which are assumed not to fail, and *unreliable* locations, which may fail. In the above example l is reliable, at which observations can be made, while the k_i are assumed unreliable, subject to failure. Furthermore it is essential that observers not have access to to these unreliable locations, at any time during a computation. Otherwise (1) would no longer represent M being fault tolerant; for example we would no longer have

$$F^1[\Gamma \triangleright \mathsf{server}_2] \cong \Gamma \triangleright \mathbf{server}_2$$

as an observer with access to k_i would be able to detect possible failures in $F^1[\Gamma \triangleright \mathsf{server}_2]$, not present in $\Gamma \triangleright \mathbf{server}_2$.

We enforce this separation between reliable, observable, locations, and unreliable, unobservable, locations, using a simple type system in which the former are called *public*, and the latter *confined*. This is outlined in Section 2, where we also formally define the language we use, DπLoc, give its reduction semantics, and also outline the behavioural equivalence $\cong$; this last is simply an instance of *reduction barbed congruence*, [6], modified so that observations can only be made at public locations. In Section 3 we give our formal definition of fault-tolerance; actually we give two versions of (1) above, called *static* and *dynamic* fault tolerance; we also motivate the difference with examples. Proof techniques for establishing fault tolerance are given in Section 4; in particular we give a complete co-induction characterisation of $\cong$, using labelled actions, and some useful up-to techniques for presenting witness bisimulations. In Section 5 we refine these proof techniques for the more demanding fault tolerant definition, *dynamic fault tolerance*, using *simulations*. Finally Section 6 outlines the main contributions of the paper and discusses future and related work.

2 The Language

We assume a set of *variables* VARS, ranged over by $x, y, z, \ldots$ and a separate set of *names*, NAMES, ranged over by $n, m, \ldots$, which is divided into locations,

Table 1. Syntax of typed DπF

Types

$\mathtt{T} ::= \mathsf{ch}_{\mathtt{v}}\langle\tilde{\mathtt{P}}\rangle \mid \mathtt{loc}^{\mathtt{s}}_{\mathtt{v}}$	*(stateful types)*	$\mathtt{s} ::= \mathtt{a} \mid \mathtt{d}$	*(status)*
$\mathtt{U} ::= \mathsf{ch}_{\mathtt{v}}\langle\tilde{\mathtt{P}}\rangle \mid \mathtt{loc}_{\mathtt{v}}$	*(stateless types)*	$\mathtt{v} ::= \mathtt{p} \mid \mathtt{c}$	*(visibility)*
$\mathtt{P} ::= \mathsf{ch}_{\mathtt{p}}\langle\tilde{\mathtt{P}}\rangle \mid \mathtt{loc}_{\mathtt{p}}$	*(public stateless types)*		

Processes

$P, Q ::= u!\langle V\rangle.P$	*(output)*	$\mid u?(X).P$	*(input)*
$\mid$ $\mathsf{if}\ v{=}u\ \mathsf{then}\ P\ \mathsf{else}\ Q$	*(matching)*	$\mid * u?(X).P$	*(replicated input)*
$\mid$ $(\nu\, n : \mathtt{T})P$	*(channel/location definition)*	$\mid \mathsf{go}\ u.P$	*(migration)*
$\mid$ $\mathbf{0}$	*(inertion)*	$\mid P\mid Q$	*(fork)*
$\mid$ $\mathsf{ping}\ u.P\ \mathsf{else}\ Q$	*(status testing)*		

Systems

$M, N, O ::= l[\![P]\!]$	*(located process)*	$\mid N\mid M$	*(parallel)*
$\mid$ $(\nu\, n : \mathtt{T})N$	*(hiding)*		

Locs, ranged over by $l, k, \ldots$ and channels, Chans, ranged over by $a, b, c, \ldots$. Finally we use $u, v, \ldots$ to range over the set of *identifiers*, consisting of either variables and names.

The syntax of our language, DπLoc, is a variation of Dπ [8] and is given in Table 1. The main syntactic category is that of *systems*, ranged over by M, N: these are essentially a collection of *located processes*, or *agents*, composed in parallel where location and channel names may be scoped to a subset of agents. The syntax for processes, P, Q, is an extension of that in Dπ: there is input and output on channels - here V is a tuple of identifiers, and X a tuple of variables, to be interpreted as a pattern - and standard forms of parallel composition, inertion, replicated input, local declarations, a test for equality between identifiers and migration. The only addition on the original Dπ is $\mathsf{ping}\ k.P\ \mathsf{else}\ Q$, which tests for the *status* of k in the style of [1, 10] and branches to P if k is alive and Q otherwise. For these terms we assume the standard notions of *free* and *bound* occurrences of both names and variables, together with the associated concepts of α-conversion and *substitution*. We also assume that systems are *closed*, that is they have no free variable occurrences.

As explained in the Introduction we use a variation (and simplification) of the type system of Dπ [8] in which the the two main categories, channels and locations, are now annotated by visibility constraints, giving $\mathsf{ch}_{\mathtt{v}}\langle\tilde{\mathtt{P}}\rangle$ and $\mathtt{loc}_{\mathtt{v}}$, where $\mathtt{v}$ may either be $\mathtt{p}$ (i.e. public) or $\mathtt{c}$ (i.e. confined); in Table 1 these are called *stateless* types, and are ranged over by $\mathtt{U}$. As explained in [5] a simple reduction semantics can be defined if we also allow types which record the status of a location, whether it is alive, $\mathtt{a}$, or dead, $\mathtt{d}$; these are refereed to as *stateful* types, ranged over by $\mathtt{T}$. Finally $\mathtt{P}$ ranges over *public* types, the types assigned to all names which are visible to observers.

Type System: Γ denotes a type environment, an unordered list of tuples assigning a single *stateful* type to names, and we write $\Gamma \vdash n : \mathtt{T}$ to mean that Γ assigns

Table 2. Typing rules for typed $D\pi$Loc

Processes

(t-out)
$$\frac{\Sigma \vdash u : \mathtt{ch}\langle\tilde{\mathtt{U}}\rangle \quad \Sigma \vdash V : \tilde{\mathtt{U}} \quad \Sigma \vdash P}{\Sigma \vdash u!\langle V\rangle.P}$$

(t-in-rep)
$$\frac{\Sigma \vdash u : \mathtt{ch}\langle\tilde{\mathtt{U}}\rangle \quad \Sigma, X : \tilde{\mathtt{U}} \vdash P}{\Sigma \vdash u?(X).P \quad \Sigma \vdash *u?(X).P}$$

(t-nw)
$$\frac{\Sigma, \tilde{n} : \tilde{\mathtt{T}} \vdash P}{\Sigma \vdash (\nu\, \tilde{n} : \tilde{\mathtt{T}})P}$$

(t-cond)
$$\frac{\Sigma \vdash u : \mathtt{U},\ v : \mathtt{U} \quad \Sigma \vdash P,\ Q}{\Sigma \vdash \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

(t-fork)
$$\frac{\Sigma \vdash P,\ Q}{\Sigma \vdash P|Q}$$

(t-axiom)
$$\frac{}{\Sigma \vdash \mathbf{0}}$$

(t-go)
$$\frac{\Sigma \vdash u : \mathtt{loc} \quad \Sigma \vdash P}{\Sigma \vdash \mathsf{go}\ u.P}$$

(t-ping)
$$\frac{\Sigma \vdash u : \mathtt{loc} \quad \Sigma \vdash P,\ Q}{\Sigma \vdash \mathsf{ping}\ u.P\ \mathsf{else}\ Q}$$

Systems

(t-rest)
$$\frac{\Gamma, \tilde{n} : \tilde{\mathtt{T}} \vdash N}{\Gamma \vdash (\nu\, \tilde{n} : \tilde{\mathtt{T}})N}$$

(t-par)
$$\frac{\Gamma \vdash N,\ M}{\Gamma \vdash N|M}$$

(t-proc)
$$\frac{\Gamma \vdash l : \mathtt{loc} \quad \Gamma \vdash P}{\Gamma \vdash l[\![P]\!]}$$

Observers

(t-obs)
$$\frac{\mathbf{pub}(\Gamma) \vdash O}{\Gamma \vdash_{\mathsf{obs}} O}$$

the type $\mathtt{T}$ to n; when it is not relevant to the discussion we will sometimes drop the various annotations on these types; for example $\Gamma \vdash n : \mathtt{ch}\langle\mathtt{U}\rangle$ signifies that $\mathtt{ch}_{\mathtt{v}}\langle\mathtt{U}\rangle$ for some visibility status $\mathtt{v}$. Typing judgements take the form $\Gamma \vdash N$ and defined by the rules in Table 2. In these rules, we use an extended form of type environment, Σ, which, in addition to names, also maps *variables* to *stateless* types. Note that none of the rules depend on the status (dead or alive) of names in the environment. Also the visibility constraints are enforced indirectly, by virtue of the formation rules for valid types, given in Table 1.

In this extended abstract we omit even the statement of the Subject Reduction and appropriate Type Safety result for our language.

Reduction Semantics: We call pairs $\Gamma \triangleright N$ *configurations*, whenever $\Gamma \vdash N$. Reductions then take the form of a binary relation over configurations

$$\Gamma \triangleright N \quad \longrightarrow \quad \Gamma \triangleright N'$$

defined in terms of the reduction rules in Table 3, whereby systems reduce with respect to the status of the locations in Γ; we write $\Gamma \vdash l : \mathbf{alive}$ as a shorthand for $\Gamma \vdash l : \mathtt{loc}^{\mathsf{a}}$. So all reduction rules assume the location where the code is executing is alive. Moreover, (r-go), (r-ngo), (r-ping) and (r-nping) reduce according to the status of the remote location concerned. The reader is refered to [5] for more details; but note that here the status of locations is unchanged by reductions.

Behavioural equivalence: First note that the type system does indeed enforce the intuitive separation of concerns discussion in the Introduction. For example let Γ_e denote the environment

Table 3. Reduction Rules for DπLoc

Assuming $\Gamma \vdash l : \textbf{alive}$

(r-comm)
$$\frac{}{\Gamma \triangleright l[\![a!\langle V\rangle.P]\!] \mid l[\![a?(X).Q]\!] \longrightarrow \Gamma \triangleright l[\![P]\!] \mid l[\![Q\{^V/_X\}]\!]}$$

(r-rep)
$$\frac{}{\Gamma \triangleright l[\![*a?(X).P]\!] \longrightarrow \Gamma \triangleright l[\![a?(X).(P \mid *a?(X).P)]\!]}$$

(r-fork)
$$\frac{}{\Gamma \triangleright l[\![P|Q]\!] \longrightarrow \Gamma \triangleright l[\![P]\!] \mid l[\![Q]\!]}$$

(r-eq)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{if}\ u{=}u\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \longrightarrow \Gamma \triangleright l[\![P]\!]}$$

(r-neq)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{if}\ u{=}v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \longrightarrow \Gamma \triangleright l[\![Q]\!]}\ u \neq v$$

(r-go)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{go}\ k.P]\!] \longrightarrow \Gamma \triangleright k[\![P]\!]}\ \Gamma \vdash k : \textbf{alive}$$

(r-ngo)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{go}\ k.P]\!] \longrightarrow \Gamma \triangleright k[\![\mathbf{0}]\!]}\ \Gamma \nvdash k : \textbf{alive}$$

(r-ping)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{ping}\ k.P\ \mathsf{else}\ Q]\!] \longrightarrow \Gamma \triangleright l[\![P]\!]}\ \Gamma \vdash k : \textbf{alive}$$

(r-nping)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{ping}\ k.P\ \mathsf{else}\ Q]\!] \longrightarrow \Gamma \triangleright l[\![Q]\!]}\ \Gamma \nvdash k : \textbf{alive}$$

(r-new)
$$\frac{}{\Gamma \triangleright l[\![(\nu n{:}\mathtt{T})P]\!] \longrightarrow \Gamma \triangleright (\nu n{:}\mathtt{T})\, l[\![P]\!]}$$

(r-str)
$$\frac{\Gamma \triangleright N' \equiv \Gamma \triangleright N \quad \Gamma \triangleright N \longrightarrow \Gamma' \triangleright M \quad \Gamma' \triangleright M \equiv \Gamma' \triangleright M'}{\Gamma \triangleright N' \longrightarrow \Gamma' \triangleright M'}$$

(r-ctxt-rest)
$$\frac{\Gamma + n : \mathtt{T} \triangleright N \longrightarrow \Gamma' + n : \mathtt{U} \triangleright M}{\Gamma \triangleright (\nu n : \mathtt{T})N \longrightarrow \Gamma' \triangleright (\nu n : \mathtt{U})M}$$

(r-ctxt-par)
$$\frac{\Gamma \triangleright N \longrightarrow \Gamma' \triangleright N'}{\begin{array}{l}\Gamma \triangleright N|M \longrightarrow \Gamma' \triangleright N'|M \\ \Gamma \triangleright M|N \longrightarrow \Gamma' \triangleright M|N'\end{array}}$$

Table 4. Structural Rules for DπLoc

(s-comm)	$N\|M \equiv M\|N$	
(s-assoc)	$(N\|M)\|M' \equiv N\|(M\|M')$	
(s-unit)	$N\|l[\![\mathbf{0}]\!] \equiv N$	
(s-extr)	$(\nu n{:}\mathtt{T})(N\|M) \equiv N\|(\nu n{:}\mathtt{T})M$	$n \notin \mathbf{fn}(N)$
(s-flip)	$(\nu n{:}\mathtt{T})(\nu m{:}\mathtt{U})N \equiv (\nu m{:}\mathtt{U})(\nu n{:}\mathtt{T})N$	
(s-inact)	$(\nu n{:}\mathtt{T})N \equiv N$	$n \notin \mathbf{fn}(N)$

$$\Gamma_e = l{:}\mathtt{loc}^{\mathtt{a}}_{\mathtt{p}},\ k_1{:}\mathtt{loc}^{\mathtt{a}}_{\mathtt{c}},\ k_2{:}\mathtt{loc}^{\mathtt{a}}_{\mathtt{c}},\ k_3{:}\mathtt{loc}^{\mathtt{a}}_{\mathtt{c}},\ req{:}\mathtt{ch_p}\langle \mathtt{T}, \mathtt{ch_p}\langle \mathtt{T}\rangle\rangle,\ a{:}\mathtt{ch_p}\langle \mathtt{A}\rangle,\ ret{:}\mathtt{ch_p}\langle \mathtt{T}\rangle$$

where $\mathtt{T}$ is an arbitrary public type; Then one can check

$$\Gamma_e \vdash \mathsf{server}_i$$

where server_i is defined in the Introduction, provided the locally declared channels *data* and sync are declared at the types $\mathtt{ch}\langle \mathtt{T}, \mathtt{ch_p}\langle \mathtt{T}\rangle, \mathtt{loc_p}\rangle$ and $\mathtt{ch}\langle \mathtt{T}\rangle$ respectively. Now consider

$$\mathsf{serverBad} \Leftarrow \mathsf{server}_1 \mid l[\![a!\langle k_1\rangle]\!]$$

which attempts to export a confined location k_1, which could subsequently could be tested for failure by a public observer. Once more one can check that $\Gamma_e \nvdash$ serverBad.

Intuitively an observer is any system which only uses public names. Formally let $\mathbf{pub}(\Gamma)$ be the environment obtained by omitting from Γ any name not assigned a public type. Then $\mathbf{pub}(\Gamma) \vdash O$ ensures that O can only use public names. For example consider

$$\begin{aligned} \mathsf{observer} &\Leftarrow l[\![req!\langle v, ret\rangle]\!] \\ \mathsf{observerBad} &\Leftarrow l[\![\mathsf{go}\ k_1.\mathsf{go}\ l.ok!\langle\rangle]\!] \end{aligned}$$

Here one can check that $\mathbf{pub}(\Gamma_e) \vdash$ observer and $\mathbf{pub}(\Gamma_e) \nvdash$ observerBad.

Our behavioural equivalence will in general relate arbitrary configurations; but we would expect equivalent configurations to have the same *public interface*, and be preserved by public observers.

Definition 1 (p-Contextual). *A relation over configurations is called p-Contextual if, whenever* $\Gamma \triangleright M \ \mathcal{R}\ \Gamma' \triangleright N$

- *(p-Interfaces:)* $\mathbf{pub}(\Gamma) = \mathbf{pub}(\Gamma')$
- *(Parallel:)* $\Gamma \triangleright M \mid O \ \mathcal{R}\ \Gamma' \triangleright N \mid O$ *and* $\Gamma \triangleright M \mid O \ \mathcal{R}\ \Gamma' \triangleright N \mid O$ *whenever* $\mathbf{pub}(\Gamma) \vdash O$
- *(Fresh extensions:)* $\Gamma, n :: \mathsf{P} \triangleright M \ \mathcal{R}\ \Gamma', n :: \mathsf{P} \triangleright N$ *whenever* n *is fresh*

Definition 2 (p-Barb). $\Gamma \triangleright N \Downarrow^p_{a@l}$ *denotes a* p-observable barb *by configuration* $\Gamma \triangleright N$*, on channel* a *at location* l*, defined as:*

$$\exists N'.\ \Gamma \triangleright N \longrightarrow^* \Gamma \triangleright N' \text{ such that } N' \equiv (\nu\,\tilde{n} : \tilde{\mathsf{T}})M | l[\![a!\langle V\rangle.Q]\!] \text{ where } \Gamma \vdash l : \mathtt{loc}^{\mathsf{a}}_{\mathsf{p}}, a : \mathsf{ch}_{\mathsf{p}}\langle\tilde{\mathsf{P}}\rangle$$

Using this concept, we can now modify the standard definition of *reduction barbed equivalence*, [6]:

Definition 3 (Reduction barbed congruence). *Let* $\cong$ *be the largest relation between configurations which is p-contextual, reduction-closed (see [6]) and preserves p-barbs.*

3 Defining Fault Tolerance

Our first notion of n-fault-tolerance, formalising the intuitive (1), is when the faulting context induces at most n location failures, prior to the execution of the system; of course these failures must only be induced on locations which are not public. Formally for any set of location names $\tilde{l}$ let $F^{\tilde{l}}_S$ be the function which maps any configuration $\Gamma \triangleright N$ to $\Gamma - \tilde{l} \triangleright N$, where $\Gamma - \tilde{l}$ is the environment obtained from Γ by changing the status of every l_i to *dead*. We say $F^{\tilde{l}}_S$ is a *valid static n-fault context* with respect to Γ, if the size of $\tilde{l}$ is *at most* n, and for every $l_i \in \tilde{l}$, l_i is confined and alive ($\Gamma \vdash l_i : \mathtt{loc}^{\mathsf{a}}_{\mathsf{c}}$).

Definition 4 (Static Fault Tolerance). *A configuration $\Gamma \triangleright N$ is n-static fault tolerant if*

$$\Gamma \triangleright N \cong F_S^{\tilde{l}}(\Gamma) \triangleright N$$

for every static n-fault context $F_S^{\tilde{l}}$ which is valid with respect to Γ.

With this formal definition we can now examine the systems server_i, using the Γ_e defined above. We can easily check that $\Gamma \triangleright \mathsf{server}_1$ is not 1-fault tolerant, by considering the fault context $F_S^{k_1}$. Similarly we can show that $\Gamma_e \triangleright \mathsf{server}_2$ is not 2-fault tolerant, by considering $F_S^{k_1,k_2}$. But establishing positive results, for example that $\Gamma_e \triangleright \mathsf{server}_2$ is 1-fault tolerant, is difficult because the definition of $\cong$ quantifies over all valid observers. This point will be addressed in the next section, when we give a co-inductive characterisation of $\cong$.

Instead let us consider another manner of inducing faults. Let $l[\![\mathsf{kill}]\!]$ be a system which asynchronously kills a confined location l. Its operation is defined by the rule

$$\text{(r-kill)}\quad \frac{}{\Gamma \triangleright l[\![\mathsf{kill}]\!] \longrightarrow (\Gamma - l) \triangleright l[\![\mathbf{0}]\!]}$$

For any set of locations $\tilde{l}$ let $F_D^{\tilde{l}}$ denote the function which maps the system M to $M \mid l_1[\![\mathsf{kill}]\!] \mid \ldots \mid l_n[\![\mathsf{kill}]\!]$. It is said to be a valid dynamic n-fault context with respect to Γ if again the size of $\tilde{l}$ is at most n and $\Gamma \vdash l_i : \mathtt{loc}_\mathsf{c}^\mathsf{a}$, for every l_i in $\tilde{l}$.

Definition 5 (Dynamic Fault Tolerance). *A configuration $\Gamma \triangleright N$ is n-dynamic fault tolerant if*

$$\Gamma \triangleright F_D^{\tilde{l}}(M) \cong \Gamma \triangleright M$$

for every dynamic n-fault context which is valid with respect to Γ.

Example 2. The system $\mathsf{sPassive}$ defined below uses two identical replicas of the distributed database at k_1 and k_2, but treats the replica at k_1 as *primary* replica and the one at k_2 as a *secondary* (backup) replica - once again $\mathtt{W} = \mathtt{ch}\langle \mathtt{T}, \mathtt{ch_p}\langle \mathtt{T}\rangle, \mathtt{loc_p}\rangle$.

$$\mathsf{sPassive} \Leftarrow (\nu\, data{:}\mathtt{W}) \left(\begin{array}{l} l \left[\!\!\left[\begin{array}{l} serv?(x,y).\mathsf{ping}\ k_1.\mathsf{go}\ k_1.data!\langle x,y,l\rangle \\ \qquad\qquad\qquad\qquad \mathsf{else\ go}\ k_2.data!\langle x,y,l\rangle \end{array} \right]\!\!\right] \\ \mid k_1[\![data?(x,y,z).\mathsf{go}\ z\ .y!\langle f(x)\rangle]\!] \\ \mid k_2[\![data?(x,y,z).\mathsf{go}\ z\ .y!\langle f(x)\rangle]\!] \end{array} \right)$$

The coordinating interface at l uses the ping construct to *detect failures* in the primary replica: if k_1 is alive, the request is sent to the primary replica and the secondary replica at k_2 is *not invoked*; if, on the other hand, the primary replica is dead, then the passive replica at k_2 is promoted to a primary replica and the request is sent to it. This implementation saves on *time redundancy* since, for any request, only one replica is invoked. Another passive replication server is $\mathsf{sMonitor}$, defined as

$$\mathsf{sMonitor} \Leftarrow (\nu\, data\!:\!\mathtt{W})\begin{pmatrix} l\left[\!\!\left[serv?(x,y).(\nu\, sync\!:\!\mathsf{ch}\langle\rangle)\begin{pmatrix}\mathsf{go}\; k_1.data!\langle x, sync, l\rangle\\ \mid \mathsf{mntr}\; k_1.\mathsf{go}\; k_2.data!\langle x, sync, l\rangle\\ \mid sync?(z).y!\langle z\rangle\end{pmatrix}\right]\!\!\right]\\ \mid k_1[\![data?(x,y,z).\mathsf{go}\; z\;.y!\langle f(x)\rangle]\!]\\ \mid k_2[\![data?(x,y,z).\mathsf{go}\; z\;.y!\langle f(x)\rangle]\!]\end{pmatrix}$$

where again, $\mathtt{W} = \mathtt{ch}\langle \mathtt{T}, \mathtt{ch_p}\langle \mathtt{T}\rangle, \mathtt{loc_p}\rangle$. It uses a *monitor* process for failure detection

$$\mathsf{mntr}\; k.P \Leftarrow (\nu\; test\!:\!\mathtt{ch}\langle\rangle)(\; test!\langle\rangle \mid *\; test?().\mathsf{ping}\; k.\; test!\langle\rangle\; \mathsf{else}\, P\;)$$

instead of a *single* ping test on the primary replica at k_1; $\mathsf{mntr}\; k.P$ *repeatedly* tests the status of the monitored location (k) and continues as P when k becomes dead. Similar to $\mathsf{server}_{2..3}$, sMonitor synchronises multiple answers from replicas with channel *sync*.

Using the techniques of the next section, one can show that both $\Gamma_e \triangleright \mathsf{sPassive}$ and $\Gamma \triangleright \mathsf{sMonitor}$, are 1-static fault tolerant, similar to server_2. However there is a difference between these two systems;if k_1 fails *after* sPassive tests for its status, then an answer will never reach l. Thus sPassive is *not* 1-dynamic fault tolerant; formally one can show $\Gamma_e \triangleright F_D^{k_1}(\mathsf{sPassive}) \not\cong \Gamma_e \triangleright \mathsf{sPassive}$. But, as we will see in the next section, sMonitor can be shown to be 1-dynamic fault tolerant, just like $\mathsf{server}_{2..3}$.

4 Proof Techniques for Fault Tolerance

We define a labelled transition system (lts) for $\mathrm{D}\pi\mathrm{Loc}_e$, which consists of a collection of actions over (closed) configurations, $\Gamma \triangleright N \xrightarrow{\mu} \Gamma' \triangleright N'$, where μ can be an internal action, τ, a bound input, $(\tilde{n} : \tilde{\mathtt{T}})l : a?(V)$ or bound output, $(\tilde{n} : \tilde{\mathtt{T}})l : a!\langle V\rangle$. These actions are defined by transition rules given in Table 5, inspired by [7, 6, 5]. In accordance with Definition 2 (observable barbs) and Definitions 1 (valid observers), (l-in) and (l-out) restrict external communication to *public* channels at *public* locations ($\Gamma \vdash_{\mathsf{obs}} l, a$). Furthermore, in (l-in) we require that the type of the values inputted, V, match the object type of channel a; since a is public and configurations are well-typed, this also implies that V are public values defined in Γ. The restriction on output action, together with the assumption of well-typed configurations also means that, in (l-open), we only scope extrude public values. Contrary to [5], the lts does not allow external killing of locations (through the label $\mathsf{kill} : l$) since public locations are reliable and never fail. Finally, the transition rule for internal communication, (l-par-comm), uses an overloaded function $\uparrow()$ for inferring input/output capabilities of the subsystems: when applied to types, $\uparrow(\mathtt{T})$ transforms all the type tags to public (p); when applied to environments, $\uparrow(\Gamma)$ changes all the types to public types in the same manner. All the remaining rules are a simplified version of the rules in [5].

Definition 6 (Weak bisimulation equivalence). *This is denoted as $\approx$, and is defined to be the largest typed relation over configurations such that if $\Gamma \triangleright M \approx \Gamma' \triangleright N$ then*

Table 5. Operational Rules for Typed DπLoc

Assuming $\Gamma \vdash l : \textbf{alive}$

(l-in)
$$\frac{}{\Gamma \triangleright l[\![a?(X).P]\!] \xrightarrow{l:a?(V)} \Gamma \triangleright l[\![P\{^V/_X\}]\!]} \; \Gamma \vdash_{\mathsf{obs}} l,\ \Gamma \vdash a : \mathrm{ch_p}\langle\tilde{\mathrm{W}}\rangle,\ V : \tilde{\mathrm{W}}$$

(l-fork)
$$\frac{}{\Gamma \triangleright l[\![P|Q]\!] \xrightarrow{\tau} \Gamma \triangleright l[\![P]\!] \mid l[\![Q]\!]}$$

(l-out)
$$\frac{}{\Gamma \triangleright l[\![a!\langle V\rangle.P]\!] \xrightarrow{l:a!\langle V\rangle} \Gamma \triangleright l[\![P]\!]} \; \Gamma \vdash_{\mathsf{obs}} l, a$$

(l-in-rep)
$$\frac{}{\Gamma \triangleright l[\![*a?(X).P]\!] \xrightarrow{\tau} \Gamma \triangleright l[\![a?(X).(P|*a?(Y).P\{^Y/_X\})]\!]}$$

(l-eq)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{if}\ u{=}u\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau} \Gamma \triangleright l[\![P]\!]}$$

(l-neq)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{if}\ u{=}v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau} \Gamma \triangleright l[\![Q]\!]} \; u \neq v$$

(l-new)
$$\frac{}{\Gamma \triangleright l[\![(\nu n : \mathrm{T})P]\!] \longrightarrow \Gamma \triangleright (\nu n : \mathrm{T})\, l[\![P]\!]}$$

(l-kill)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{kill}]\!] \xrightarrow{\tau} (\Gamma - l) \triangleright l[\![\mathbf{0}]\!]}$$

(l-go)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{go}\ k.P]\!] \xrightarrow{\tau} \Gamma \triangleright k[\![P]\!]} \; \Gamma \vdash k : \textbf{alive}$$

(l-ngo)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{go}\ k.P]\!] \xrightarrow{\tau} \Gamma \triangleright k[\![\mathbf{0}]\!]} \; \Gamma \nvdash k : \textbf{alive}$$

(l-ping)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{ping}\ k.P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau} \Gamma \triangleright l[\![P]\!]} \; \Gamma \vdash k : \textbf{alive}$$

(l-nping)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{ping}\ k.P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau} \Gamma \triangleright l[\![Q]\!]} \; \Gamma \nvdash k : \textbf{alive}$$

(l-open)
$$\frac{\Gamma + n : \mathrm{T} \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathrm{T}})l:a!\langle V\rangle} \Gamma' \triangleright N'}{\Gamma \triangleright (\nu n : \mathrm{T})N \xrightarrow{(n:\mathrm{T},\tilde{n}:\tilde{\mathrm{T}})l:a!\langle V\rangle} \Gamma' \triangleright N'} \; l, a \neq n \in V$$

(l-weak)
$$\frac{\Gamma + n : \mathrm{T} \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathrm{T}})l:a?(V)} \Gamma' \triangleright N'}{\Gamma \triangleright N \xrightarrow{(n:\mathrm{T},\tilde{n}:\tilde{\mathrm{T}})l:a?(V)} \Gamma' \triangleright N'} \; l, a \neq n \in V$$

(l-rest)
$$\frac{\Gamma + n : \mathrm{T} \triangleright N \xrightarrow{\mu} \Gamma' + n : \mathrm{U} \triangleright N'}{\Gamma \triangleright (\nu n : \mathrm{T})N \xrightarrow{\mu} \Gamma' \triangleright (\nu n : \mathrm{U})N'} \; n \notin \mathbf{fn}(\mu)$$

(l-par-ctxt)
$$\frac{\Gamma \triangleright N \xrightarrow{\mu} \Gamma' \triangleright N'}{\begin{array}{c}\Gamma \triangleright N \mid M \xrightarrow{\mu} \Gamma' \triangleright N' \mid M \\ \Gamma \triangleright M \mid N \xrightarrow{\mu} \Gamma' \triangleright M \mid N'\end{array}}$$

(l-par-comm)
$$\frac{\uparrow(\Gamma) \triangleright N \xrightarrow{(\tilde{n}:\uparrow(\tilde{\mathrm{T}}))l:a!\langle V\rangle} \Gamma' \triangleright N' \qquad \uparrow(\Gamma) \triangleright M \xrightarrow{(\tilde{n}:\uparrow(\tilde{\mathrm{T}}))l:a?(V)} \Gamma'' \triangleright M'}{\begin{array}{c}\Gamma \triangleright N \mid M \xrightarrow{\tau} \Gamma \triangleright (\nu \tilde{n} : \tilde{\mathrm{T}})(N' \mid M') \\ \Gamma \triangleright M \mid N \xrightarrow{\tau} \Gamma \triangleright (\nu \tilde{n} : \tilde{\mathrm{T}})(M' \mid N')\end{array}}$$

– $\Gamma \triangleright M \xrightarrow{\mu} \Gamma'' \triangleright M'$ *implies* $\Gamma' \triangleright N \overset{\hat{\mu}}{\Longrightarrow} \Gamma''' \triangleright N'$ *such that* $\Gamma'' \triangleright M' \approx \Gamma''' \triangleright N'$
– $\Gamma' \triangleright N \xrightarrow{\mu} \Gamma'' \triangleright N'$ *implies* $\Gamma \triangleright M \overset{\hat{\mu}}{\Longrightarrow} \Gamma'' \triangleright M'$ *such that* $\Gamma'' \triangleright M' \approx \Gamma''' \triangleright N'$

Theorem 1 (Full Abstraction). *For any DπLoc configurations $\Gamma \triangleright M$, $\Gamma' \triangleright N$:*

$$\Gamma \triangleright M \cong \Gamma' \triangleright N \text{ if and only if } \Gamma \triangleright M \approx \Gamma' \triangleright N$$

Table 6. β-Transition Rules for Typed DπLoc

Assuming $\Gamma \vdash l : \mathbf{alive}$

(b-in-rep)
$$\frac{}{\Gamma \triangleright l[\![*a?(X).P]\!] \overset{\tau}{\longmapsto}_\beta \Gamma \triangleright l[\![a?(X).(P \mid *a?(Y).P\{^Y/_X\})]\!]}$$

(b-fork)
$$\frac{}{\Gamma \triangleright l[\![P|Q]\!] \overset{\tau}{\longmapsto}_\beta \Gamma \triangleright l[\![P]\!] \mid l[\![Q]\!]}$$

(b-eq)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{if}\ u = u\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \overset{\tau}{\longmapsto}_\beta \Gamma \triangleright l[\![P]\!]}$$

(b-neq)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \overset{\tau}{\longmapsto}_\beta \Gamma \triangleright l[\![Q]\!]}\ u \neq v$$

(b-ngo)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{go}\ k.P]\!] \overset{\tau}{\longmapsto}_\beta \Gamma \triangleright k[\![\mathbf{0}]\!]}\ \Gamma \nvdash k : \mathbf{alive}$$

(b-nping)
$$\frac{}{\Gamma \triangleright l[\![\mathsf{ping}\ k.P\ \mathsf{else}\ Q]\!] \overset{\tau}{\longmapsto}_\beta \Gamma \triangleright l[\![Q]\!]}\ \Gamma \nvdash k : \mathbf{alive}$$

(b-new)
$$\frac{}{\Gamma \triangleright l[\![(\nu n : \mathtt{T})P]\!] \overset{\tau}{\longmapsto}_\beta \Gamma \triangleright (\nu n : \mathtt{T}) l[\![P]\!]}$$

(b-rest)
$$\frac{\Gamma, n : \mathtt{T} \triangleright N \overset{\tau}{\longmapsto}_\beta \Gamma', n : \mathtt{W} \triangleright N'}{\Gamma \triangleright (\nu n : \mathtt{T}) N \overset{\tau}{\longmapsto}_\beta \Gamma' \triangleright (\nu n : \mathtt{W}) N'}$$

(b-par)
$$\frac{\Gamma \triangleright N \overset{\tau}{\longmapsto}_\beta \Gamma' \triangleright N'}{\begin{array}{c}\Gamma \triangleright N|M \overset{\tau}{\longmapsto}_\beta \Gamma' \triangleright N'|M \\ \Gamma \triangleright M|N \overset{\tau}{\longmapsto}_\beta \Gamma' \triangleright M|N'\end{array}}$$

Theorem 1 allows us to prove *positive* fault tolerance results by giving a bisimulation for every reduction barbed congruent pair required by Definitions 4 and 5. We next develop up-to bisimulation techniques that can relieve some of the burden of exhibiting the required bisimulations. We identify a number of τ actions, which we refer to as β-actions or β-moves, inspired by the work in [3]. These are denoted as $\Gamma \triangleright N \overset{\tau}{\longmapsto}_\beta \Gamma' \triangleright N$ and are defined in Table 6. With these β-moves we develop up-to bisimulation techniques, by showing that our witness bisimulations can abstract away from matching configurations that denote β-moves. Our details are more complicated than in [3] because we deal with failure: apart from local rules ((b-eq) and (b-fork)) and context rules ((b-rest) and (b-par)), Table 6 includes rules dealing with failed locations such as (b-ngo) and (b-nping). To obtain the required results for β-moves with failure, we define a new structural equivalence ranging over *configurations*, denoted as $\equiv_f$ and defined by the rules in Table 7, which takes into consideration *location status* as well. This enables us to obtain confluence for β-moves with respect to actions that change the status of locations. The only rule worth highlighting is (bs-dead), which allows us to ignore dead code.

Lemma 1 (Confluence of β-moves). $\overset{\tau}{\longmapsto}_\beta$ *observes the diamond property:*

$$\begin{array}{ccccccc}
\Gamma \triangleright N & \overset{\tau}{\longmapsto}_\beta & \Gamma \triangleright M & \quad \textit{implies} \quad & \Gamma \triangleright N & \overset{\tau}{\longmapsto}_\beta & \Gamma \triangleright M \\
\downarrow \mu & & & & \downarrow \mu & & \downarrow \mu \\
\Gamma' \triangleright N' & & \Gamma' \triangleright M' & & \Gamma' \triangleright N' & \overset{\tau}{\Longmapsto}_\beta \equiv_f & \Gamma' \triangleright M'
\end{array} \quad \textit{or } \mu = \tau \textit{ and } \Gamma \triangleright M = \Gamma' \triangleright N'$$

Table 7. β-Equivalence Rules for Typed DπLoc

(bs-comm)	$\Gamma \models N\|M \equiv_f M\|N$	
(bs-assoc)	$\Gamma \models (N\|M)\|M' \equiv_f N\|(M\|M')$	
(bs-unit)	$\Gamma \models N\|l[\![\mathbf{0}]\!] \equiv_f N$	
(bs-extr)	$\Gamma \models (\nu n:\mathtt{T})(N\|M) \equiv_f N\|(\nu n:\mathtt{T})M$	$n \notin \mathbf{fn}(N)$
(bs-flip)	$\Gamma \models (\nu n:\mathtt{T})(\nu m:\mathtt{U})N \equiv_f (\nu m:\mathtt{U})(\nu n:\mathtt{T})N$	
(bs-inact)	$\Gamma \models (\nu n:\mathtt{T})N \equiv_f N$	$n \notin \mathbf{fn}(N)$
(bs-dead)	$\Gamma \models l[\![P]\!] \equiv_f l[\![Q]\!]$	$\Gamma \nvdash l : \mathbf{alive}$

Proof. The proof proceeds by case analysis of the different types of μ and then by induction on the derivation of the β-move.

Proposition 1. *Suppose $\Gamma \triangleright N \models\!\!\Longrightarrow_\beta \Gamma' \triangleright M$. Then $\Gamma \triangleright N \approx \Gamma' \triangleright M$.*

Proof. We prove the above statement by defining $\mathcal{R} = \{\Gamma \triangleright N, \Gamma' \triangleright M \mid \Gamma \triangleright N \models\!\!\Longrightarrow_\beta \Gamma' \triangleright M\}$ and showing that $\mathcal{R}$ is a bisimulation, which follows as a consequence of Lemma 1.

Definition 7 (Bisimulation up-to β-moves). *Bisimulation up-to β-moves, denoted as $\approx_\beta$, is the largest typed relation between configurations such that $\Gamma_1 \triangleright M_1 \approx_\beta \Gamma_2 \triangleright M_2$ and*

- *$\Gamma_1 \triangleright M_1 \xrightarrow{\mu} \Gamma_1' \triangleright M_1'$ implies $\Gamma_2 \triangleright M_2 \overset{\hat{\mu}}{\Longrightarrow} \Gamma_2' \triangleright M_2'$ such that $\Gamma_1' \triangleright M_1' \, \mathcal{A}_l \circ \approx_\beta \circ \approx \Gamma_2' \triangleright M_2'$*
- *$\Gamma_2 \triangleright M_2 \xrightarrow{\mu} \Gamma_2' \triangleright M_2'$ implies $\Gamma_1 \triangleright M_1 \overset{\hat{\mu}}{\Longrightarrow} \Gamma_1' \triangleright M_1'$ such that $\Gamma_2' \triangleright M_2' \, \mathcal{A}_l \circ \approx_\beta \circ \approx \Gamma_1' \triangleright M_1'$*

where $\mathcal{A}_l$ is the relation $\models\!\!\Longrightarrow_\beta \circ \equiv$.

Proposition 1 provides us with a powerful method for approximating bisimulations. In the approximate bisimulation $\approx_\beta$, an action $\Gamma_1 \triangleright M_1 \xrightarrow{\mu} \Gamma_1' \triangleright M_1'$ can be matched by a β-derivative of $\Gamma_1' \triangleright M_1'$, that is $\Gamma_1' \triangleright M_1' \models\!\!\Longrightarrow_\beta \Gamma_1' \triangleright M_1''$, and a weak matching action $\Gamma_2 \triangleright M_2 \overset{\hat{\mu}}{\Longrightarrow} \Gamma_2' \triangleright M_2'$ such that, up to structural equivalence on the one side and up-to bisimilarity on the other, the pairs $\Gamma_1' \triangleright M_1''$ and $\Gamma_2' \triangleright M_2'$ are once more related. Intuitively then, in any relation satisfying $\approx_\beta$, a configuration can represent all the configurations to which it can evolve using β-moves. We justify the use of $\approx_\beta$ by proving Proposition 2.

Proposition 2 (Inclusion of bisimulation up-to β-moves). *If $\Gamma_1 \triangleright M_1 \approx_\beta \Gamma_2 \triangleright M_2$ then $\Gamma_1 \triangleright M_1 \approx \Gamma_2 \triangleright M_2$*

Proof. We prove the above proposition by defining the relation $\mathcal{R}$ as

$$\mathcal{R} = \left\{ \Gamma_1 \triangleright M_1 \, , \Gamma_2 \triangleright M_2 \quad \middle| \Gamma_1 \triangleright M_1 \approx \circ \approx_\beta \circ \approx \Gamma_2 \triangleright M_2 \right\}$$

and show that $\mathcal{R} \subseteq \approx$. The required result can then be extracted from this result by considering the special cases where the $\approx$ on either side are the identity relations.

Example 3. We are now in a position to prove positive fault tolerance result. For instance to show that $\Gamma \triangleright \mathsf{sPassive}$ is 1-static fault tolerant we just need to provide 3 witness bisimulations up-to β-moves to prove

$$\prod_{i=1}^{3} \ \Gamma \triangleright \mathsf{sPassive} \ \cong \ (\Gamma - k_i) \triangleright \mathsf{sPassive}$$

We here give the witness relation for the most involving case (where $i = 1$), and leave the simpler relations for the interested reader. Thus, the witness relation is $\mathcal{R}$ defined as

$$\mathcal{R} \stackrel{\text{def}}{=} \{\langle \Gamma \triangleright \mathsf{sPassive}, \Gamma - k_1 \triangleright \mathsf{sPassive}\rangle\} \ \cup \ \left(\bigcup_{u,v \in \text{NAMES}} \mathcal{R}'(u,v) \right)$$

$$\mathcal{R}'(u,v) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \Gamma \triangleright (\nu d) l[\![\mathsf{Png}(u,v)]\!] \mid \mathsf{R}_1 \mid \mathsf{R}_2 & , \Gamma - k_1 \triangleright (\nu d) l[\![\mathsf{Png}(u,v)]\!] \mid \mathsf{R}_1 \mid \mathsf{R}_2 \\ \Gamma \triangleright (\nu d) l[\![\mathsf{Q}_1(u,v)]\!] \mid \mathsf{R}_1 \mid \mathsf{R}_2 & , \Gamma - k_1 \triangleright (\nu d) l[\![\mathsf{Q}_2(u,v)]\!] \mid \mathsf{R}_1 \mid \mathsf{R}_2 \\ \Gamma \triangleright (\nu d) k_1[\![d!\langle u,v,l\rangle]\!] \mid \mathsf{R}_1 \mid \mathsf{R}_2 & , \Gamma - k_1 \triangleright (\nu d) k_2[\![d!\langle u,v,l\rangle]\!] \mid \mathsf{R}_1 \mid \mathsf{R}_2 \\ \Gamma \triangleright (\nu d) k_1[\![\mathsf{go}\, l.v!\langle f(u)\rangle]\!] \mid \mathsf{R}_2 & , \Gamma - k_1 \triangleright (\nu d) \mathsf{R}_1 \mid k_2[\![\mathsf{go}\, l.v!\langle f(u)\rangle]\!] \\ \Gamma \triangleright (\nu d) l[\![v!\langle f(u)\rangle]\!] \mid \mathsf{R}_2 & , \Gamma - k_1 \triangleright (\nu d) \mathsf{R}_1 \mid l[\![v!\langle f(u)\rangle]\!] \\ \Gamma \triangleright (\nu d) \mathsf{R}_2 & , \Gamma - k_1 \triangleright (\nu d) \mathsf{R}_1 \end{array} \right\}$$

where d stands for *data* and

$$\begin{aligned} \mathsf{Png}(x,y) &\Leftarrow \mathsf{ping}\ k_1.\mathsf{Q}_1(x,y)\ \mathsf{else}\ \mathsf{Q}_2(x,y) \\ \mathsf{Q}_i(x,y) &\Leftarrow \mathsf{go}\ k_i.d!\langle x,y,l\rangle \\ \mathsf{R}_i &\Leftarrow k_i[\![d?(x,y,z).\mathsf{go}\ z\ .y!\langle f(x)\rangle]\!] \end{aligned}$$

5 Generic Techniques for Dynamic Fault Tolerance

Despite the fault tolerance proof techniques developed in Section 4, proving positive fault tolerance results entails a lot of unnecessary repeated work because Definition 4 and Definition 5 quantify over all valid fault contexts: to prove that server_3 is 2-dynamic fault tolerant, we need to provide 6 relations, one for every different case in

$$\prod_{i \neq j=1}^{3} \ \Gamma \triangleright \mathsf{server}_3 \ \cong \ \Gamma \triangleright \mathsf{server}_3 | k_i[\![\mathsf{kill}]\!] | k_j[\![\mathsf{kill}]\!]$$

A closer inspection of the required relations reveals that there is a lot of overlap between them: these overlapping states would be automatically circumvented if we require a single relation that is somewhat the merging of all of these separate relations. Hence we reformulate our fault tolerance definition for dynamic fault tolerance (the most demanding), to reflect such a merging of relations; a similar definition for the static case should not be more difficult to construct. The new definition is based on the actions given earlier in Section 4 together with a new action, fail, defined as

$$\text{(l-fail)} \quad \frac{}{\Gamma \triangleright N \xrightarrow{\mathsf{fail}} (\Gamma - l) \triangleright N} \ \Gamma \vdash l : \mathtt{loc}_{\mathsf{c}}^{\mathsf{a}}$$

permitting external killing of confined locations. Intuitively, this action allow us to *count* the number of failures, but prohibits us from determining which specific location failed.[1] The asymmetric relation $\preceq_D^n$, defined below, is parameterised with an integer n, denoting the number of confined locations that can still be killed on the right hand side: the additional third clause states that a fail move on the right hand side may be matched by a weak τ-move on the left hand side and the two residuals need to be related in $\preceq_D^{n-1}$.

Definition 8 (Dynamic Fault Tolerance Simulation). *Dynamic n-fault tolerant* simulation*, denoted* $\preceq_D^n$*, is the largest* asymmetric *typed relation over configurations such that whenever* $\Gamma_1 \triangleright M_1 \preceq_D^n \Gamma_2 \triangleright M_2$*,*

- $\Gamma_1 \triangleright M_1 \xrightarrow{\gamma} \Gamma_1' \triangleright M_1'$ *implies* $\Gamma_2 \triangleright M_2 \xRightarrow{\hat{\gamma}} \Gamma_2' \triangleright M_2'$ *such that* $\Gamma_1' \triangleright M_1' \preceq_D^n \Gamma_2' \triangleright M_2'$
- $\Gamma_2 \triangleright M_2 \xrightarrow{\gamma} \Gamma_2' \triangleright M_2'$ *implies* $\Gamma_1 \triangleright M_1 \xRightarrow{\hat{\gamma}} \Gamma_1' \triangleright M_1'$ *such that* $\Gamma_1' \triangleright N_1' \preceq_D^n \Gamma_2' \triangleright M_2'$
- *if* $n > 0$, $\Gamma_2 \triangleright M_2 \xrightarrow{\mathsf{fail}} \Gamma_2' \triangleright M_2'$ *implies* $\Gamma_1 \triangleright M_1 \Longrightarrow \Gamma_1' \triangleright M_1'$ *such that* $\Gamma_1' \triangleright M_1' \preceq_D^{n-1} \Gamma_2' \triangleright M_2'$

Before we can use Definition 8 to prove dynamic fault tolerance, we need to show that the new definition is sound with respect to Definition 5.

Proposition 3 (Soundness of $\preceq_D^n$). *If* $\Gamma \models M_1 \preceq_D^n M_2$ *then for any dynamic n-fault context* $F_D^{\tilde{l}}$ *that is valid with respect to* Γ *we have* $\Gamma \models M_1 \cong F_D^{\tilde{l}}(M_2)$

Proof. Let $\mathcal{R}_n$ be a relation parameterised by a number n and defined as

$$\mathcal{R}_n \stackrel{\text{def}}{=} \left\{ \Gamma_1 \triangleright M_1 \;, \Gamma_2 \triangleright M_2 \,|\, F_D^i \quad \Big| \Gamma_1 \triangleright M_1 \preceq_D^i \Gamma_2 \triangleright M_2,\ \textstyle\prod_{j=1}^{2} \Gamma_j \vdash F_D^i \text{ and } 0 \leq i \leq n \right\}$$

By showing $\mathcal{R}_n \subseteq \approx$ we prove that $\preceq_D^n$ is sound with respect to n-dynamic fault tolerance

It would be ideal if we could reuse up-to techniques and give relations satisfying $\preceq_D^n$ that abstract away from β-moves. Similar to Section 4, we define a fault tolerance simulation up-to β-moves and show that this is sound with respect to $\preceq_D^n$. This definition uses a weak bisimulation (Definition 6) that ranges over α actions, that is μ *and* the new action fail. We refer to this bisimulation as a *counting* bisimulation over configurations, denoted as $\approx_{cnt}$, because it allows us to count failing confined locations on each side and match subsequent observable behaviour.

Definition 9 (Fault Tolerant Simulation up-to β-moves). *An n-fault tolerant simulation up-to β-moves, denoted as* $\preceq_\beta^n$*, is the largest typed relation* $\mathcal{R}$ *between configurations parameterised by the number n, such that whenever we have* $\Gamma_1 \triangleright M_1 \preceq_\beta^n \Gamma_2 \triangleright M_2$

- $\Gamma_1 \triangleright M_1 \xrightarrow{\mu} \Gamma_1' \triangleright M_1'$ *implies* $\Gamma_2 \triangleright M_2 \xRightarrow{\hat{\mu}} \Gamma_2' \triangleright M_2'$ *such that* $\Gamma_1' \triangleright M_1' \mathcal{A}_l \circ \preceq_\beta^n \circ \approx_{cnt} \Gamma_2' \triangleright M_2'$

[1] This point differs from [5], where labels for external killing carried the location name, kill:l.

– $\Gamma_2 \triangleright M_2 \xrightarrow{\mu} \Gamma_2' \triangleright M_2'$ *implies* $\Gamma_1 \triangleright M_1 \xRightarrow{\hat{\mu}} \Gamma_1' \triangleright M_1'$ *such that* $\Gamma_2' \triangleright M_2' \, \mathcal{A}_l \circ \preceq_\beta^n \circ \approx \Gamma_1' \triangleright M_1'$
– *If* $n > 0$ *then* $\Gamma_2 \triangleright M_2 \xrightarrow{\mathsf{fail}} \Gamma_2' \triangleright M_2'$ *implies* $\Gamma_1 \triangleright M_1 \Longrightarrow \Gamma_1' \triangleright M_1'$ *such that* $\Gamma_2' \triangleright M_2' \preceq_\beta^{n-1} \circ \approx \Gamma_1' \triangleright M_1'$

where $\mathcal{A}_l$ *is the relation* $\models\!\Longrightarrow_\beta \circ \equiv$. *We highlight the use of* $\approx_{cnt}$ *for matching configurations in the first clause.*

The work required to show that $\preceq_\beta^n$ is sound with respect to $\preceq_D^n$ is similar to earlier up-to β-moves work discussed in Section 4: we have to show that β-move confluence (similar to Lemma 1) is also preserved for the new action fail; we also have to show that after a β-move, the redex and reduct configurations are counting-bisimilar (similar to Proposition 1). Finally we prove the following proposition

Proposition 4 (Inclusion of fault tolerant simulation up-to β-moves). *If* $\Gamma_1 \triangleright M_1 \preceq_\beta^n \Gamma_2 \triangleright M_2$ *then* $\Gamma_1 \triangleright M_1 \preceq_D^n \Gamma_2 \triangleright M_2$

Proof. We prove the above proposition by defining the relation $\mathcal{R}_n$ as

$$\mathcal{R}_n = \left\{ \Gamma_1 \triangleright M_1 \,, \Gamma_2 \triangleright M_2 \quad \middle| \Gamma_1 \triangleright M_1 \approx \circ \preceq_\beta^i \circ \approx_{cnt} \Gamma_2 \triangleright M_2 \text{ and } 0 \le i \le n \right\}$$

and show that $\mathcal{R}_n \subseteq \preceq_D^n$. The required result can then be extracted from this result by considering the special cases where $\approx$ and $\approx_{cnt}$ on either side are the identity relations.

Example 4. The results of Proposition 3 and Proposition 4 allow us to prove that the configuration $\Gamma \triangleright \mathsf{server}_2$ is 1-dynamically fault tolerant by providing a *single* witness fault tolerance simulation up-to β-moves showing that $\Gamma \triangleright \mathsf{server}_2 \preceq_\beta^1 \Gamma \triangleright \mathsf{server}_2$ Due to lack of space, we relegate the presentation of this relation to the full paper [4].

6 Conclusions and Related Work

We adopted a subset of [5] and developed a theory for system fault tolerance in the presence of fail-stop node failure. We formalised two definitions for fault tolerance based on the well studied concept of observational equivalence. Subsequently, we developed various sound proof techniques with respect to these definitions.

Future Work. The immediate next step is to apply the theory to a wider spectrum of examples, namely using replicas with state and fault tolerance techniques such as lazy replication: we postulate that the existing theory should suffice. Another avenue worth considering is extending the theory to deal with link failure and the interplay between node and link failure [5]. In the long run, we plan to develop of a compositional theory of fault tolerance, enabling the construction of fault tolerant systems from smaller component sub-systems. For both cases, this paper should provide a good starting point.

Related Work. To the best of our knowledge, Prasad's thesis [9] is the closest work to ours, addressing fault tolerance for process calculi. Even though similar concepts such as redundancy (called "duplication") and failure-free execution are identified, the setting and development of Prasad differs considerably form ours. In essence, three new operators ("displace", "audit" and "checkpoint") are introduced in a CCS variant; equational laws for terms using these operators are then developed so that algebraic manipulation can be used to show that terms in this calculus are, in some sense, fault tolerant with respect to their specification.

References

1. Roberto M. Amadio and Sanjiva Prasad. Localities and failures. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 14, 1994.
2. Flavin Christian. Understanding fault tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.
3. Alberto Ciaffaglione, Matthew Hennessy, and Julian Rathke. Proof methodologies for behavioural equivalence in Dπ. Technical Report 03/2005, University of Sussex, 2005.
4. Adrian Francalanza and Matthew Hennessy. A theory for observational fault tolerance. `www.cs.um.edu.mt/~ afran/`.
5. Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failures. In *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2005.
6. Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. *Theoretical Computer Science*, 322:615–669, 2004.
7. Matthew Hennessy and Julian Rathke. Typed behavioural equivalences for processes in the presence of subtyping. *Mathematical Structures in Computer Science*, 14:651–684, 2004.
8. Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.
9. K. V. S. Prasad. *Combinators and Bisimulation Proofs for Restartable Systems.* PhD thesis, Department of Computer Science, University of Edinburgh, December 1987.
10. James Riely and Matthew Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 226:693–735, 2001.
11. Davide Sangiorgi and David Walker. *The π-calculus.* Cambridge University Press, 2001.
12. Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3):222–238, 1983.
13. Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, 2001.

Smooth Orchestrators*

Cosimo Laneve[1] and Luca Padovani[2]

[1] Department of Computer Science, University of Bologna
laneve@cs.unibo.it
[2] Information Science and Technology Institute, University of Urbino
padovani@sti.uniurb.it

Abstract. A *smooth orchestrator* is a process with several alternative branches, every one defining synchronizations among co-located channels. Smooth orchestrators constitute a basic mechanism that may express standard workflow patterns in Web services as well as common synchronization constructs in programming languages. Smooth orchestrators may be created in one location and migrated to a different one, still not manifesting problems that usually afflict generic mobile agents.

We encode an extension of Milner's (asynchronous) pi calculus with join patterns into a calculus of smooth orchestrators and we yield a strong correctness result (full abstraction) when the subjects of the join patterns are co-located. We also study the translation of smooth orchestrators into finite-state automata, therefore addressing the implementation of co-location constraints and the case when synchronizations are not linear with respect to subjects.

1 Introduction

Web services programming languages use mechanisms for defining services available on the Web. Examples of these languages are Microsoft XLANG [13] and its visual environment BizTalk, IBM WSFL [10], BPEL [2], WS-CDL [8], and WSCI [8]. Among the basic mechanisms used by such technologies, there are the so-called *orchestrators*, which compose available services, possibly located at different administrative domains, by adding a central coordinator that is responsible for invoking and combining sub-activities.

This contribution addresses a very simple class of orchestrators, those triggering a continuation when a pattern of messages on a set of services is available. For example, the orchestrator

$$x(u) \mathbin{\&} y(v) \triangleright \overline{z}\, uv$$

enables the continuation $\overline{z}\, uv$ if one message to the service x and one message to the service y are available. The orchestrator expires once it has executed. This process is easy to implement if the two services x and y – called *channels* in the following – are co-located: it suffices to migrate $x(u) \mathbin{\&} y(v) \triangleright \overline{z}\, uv$ to the location of x and y. The general case when the continuation $\overline{z}\, uv$ is a large process may be always reduced to the simpler one.

* Aspects of this investigation were partly supported by a Microsoft initiative in concurrent computing and Web services.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 32–46, 2006.
© Springer-Verlag Berlin Heidelberg 2006

If x and y are not co-located then we immediately face a *global consensus problem*: the location running the channel(-manager)s for x and y must agree with the one running $x(u) \,\&\, y(v) \triangleright \overline{z}\, uv$ for consuming outputs. (Migrating $x(u) \,\&\, y(v) \triangleright P$, or a variant of it, to the location of x or of y does not simplify the problem because x and y are not co-located.) Observe that similar problems are manifested by orchestrators such as

$$x(u) \triangleright \overline{z}\, u \quad + \quad y(v) \triangleright \overline{z'}\, v$$

where "+" picks either one of $x(u) \triangleright \overline{z}\, u$ or $y(v) \triangleright \overline{z'}\, v$ according to the availability of a message on x or on y.

A language with orchestrators should therefore simply disallow those ones that combine not co-located channels, on the grounds that they are un-implementable. There are several ways for removing such problematic orchestrators. The join calculus [6] achieves (co-)locality with an elegant syntactic constraint in which the same language construct is used both to declare channels and to define their continuations. Channels that are being orchestrated are, by definition, co-located. For example, the process

$$\begin{aligned} x(u) \triangleright (y, z)(\ & \overline{s}\, uy \mid \overline{t}\, uz \\ & \mid y(v) \triangleright P + z(w) \triangleright Q) \end{aligned}$$

specifies a service x that starts the sub-activities s and t with local channels y and z, respectively. The orchestrator – defined on these local channels – takes into account the first activity that completes. It is worth to remark that the above process remains implementable even if y and z are not co-located with x (this may be the case when, for load-balancing reasons, it is preferable to create the two channels remotely). To formalize this constraint on y and z it suffices to add an explicit co-location construct to channel definitions. Write $z \,@\, y$ to mean that z is created at the same location as y. Then the above process may be rewritten as $x(u) \triangleright (y \,@\, y, z \,@\, y)(\overline{s}\, uy \mid \overline{t}\, uz \mid y(v) \triangleright P + z(w) \triangleright Q)$, where the constraint $y \,@\, y$ means that y may be created at whatever location.

We also consider a further relaxation of the join calculus locality constraint, which combines co-location and input capability. Input capability is the ability to receive a channel name and subsequently accept inputs on it. Orchestrators that join received channels and locally defined ones are again implementable if all the channels are co-located. In this case, the co-location constraint may be enforced statically if the language has mechanisms for extracting the location information out of received channels. In practice this is trivial because channels contains the IP addresses of their location. Technically we write $x(u \,@\, u) \,\&\, y(v \,@\, u) \triangleright P$ to select messages on x and y that carry co-located channels. In this case, the continuation P could orchestrate u and v, that is P might be $u(u' \,@\, u') \,\&\, v(v' \,@\, v') \triangleright P'$.

We end up in considering a class of orchestrators, which we call *smooth* (the terminology is drawn from [1]), that consist of several alternative branches, every one defining synchronizations among co-located channels and having an output as continuation. The implementation of (smooth) orchestrators poses a number of challenges because they may be dynamically created and because of the co-location constraints that may introduce dependencies between different channels in the same join pattern. With respect to Le Fessant and Maranget's compilation technique of join patterns [9]

we discuss a number of extensions for implementing smooth orchestrators of increasing complexity. In particular, we show that it is still possible to use finite state automata for handling join pattern definitions, even when the joined channels are not fresh.

Related work. Distributed implementations of input-guarded choices have already been studied in detail by the pi community. Nestmann and Pierce have proposed the following encoding of the orchestrator $\sum_{i\in 1..n} x_i(u_i)\triangleright P$ (we rewrite the solution in our notation):

$$(\ell \,@\, \ell)(\overline{\ell}\, t \mid \prod_{i\in 1..n} x_i(u_i \,@\, u_i) \triangleright \Big(\ell(u \,@\, t) \triangleright (\overline{\ell}\, f \mid P) \;\; + \;\; \ell(u \,@\, f) \triangleright (\overline{\ell}\, f \mid \overline{x_i}\, u_i)\Big)$$

where t and f are two free channels that are not co-located. While this technique may be refined so that every branch of the choice inputs on different channels ℓ (*c.f.* linear forwarders [7]), it seems not useful for our orchestrators where guards are complex input patterns. Implementations of "defined once"-orchestrators on co-located channels have been studied in detail for the join calculus in [9]. There are similarities between our calculus and MAGNETs [3]. In MAGNETs orchestrators are implemented as agents that migrate to the location where the synchronized channels are defined. As in this paper, MAGNETs only synchronize co-located channels, even though this condition is not enforced by a type system.

The calculus of orchestrators that we study is actually intermediate between pi calculus [11] and join calculus [6]. Its motivations are pretty practical. We have recently developed a distributed machine for the pi calculus – `PiDuce` [5, 4] – where it is possible to create channels and their managers in remote locations. This machine supports inputs on received channels by decoupling them into a particle migrating to the remote channel (the linear forwarder) and the continuation. This contribution analyzes an extension of this feature with patterns of inputs and discusses the technical problems we found in prototyping them. Smooth orchestrators that coordinate local channels have been already implemented in the current `PiDuce` prototype [4]. In the next release we expect to extend the prototype with migrating smooth orchestrators and co-location constraints.

Plan of paper. The paper is structured as follows. Section 2 gives the calculus with orchestrators, and its reference semantics – barbed congruence. Section 3 gives the encoding of few sample workflow patterns. Section 4 gives the smoothness constraint on orchestrators that makes them implementable. We demonstrate the invariance of the constraint with respect to the reduction and the implementation of the full calculus. Section 5 describes the implementation of smooth orchestrators.

2 Processes with Orchestrators

In this section we introduce the calculus with orchestrators. We first present the syntax, then the co-location relation, which is preparatory to the operational semantics, and finally the operational semantics.

2.1 Syntax

We assume an infinite set of *names* ranged over by x, u, v, ... Names represent communication channels, which are also the values being transmitted in communications. We write $\widetilde{x}$ for a (possibly empty) finite sequence $x_1 \cdots x_n$ of names. *Name substitutions* $\{\widetilde{y}/\widetilde{x}\}$ are as usual and ranged over ρ, ρ'. We let $\mathtt{dom}(\{\widetilde{y}/\widetilde{x}\}) = \widetilde{x}$. We also write $(x_1, \cdots, x_n \,@\, y_1, \cdots, y_n)$ for $(x_1 \,@\, y_1) \cdots (x_n \,@\, y_n)$. These sequences, called *co-location sequences*, are ranged over by Λ, Λ'.

The syntax consists of *processes* P and *join patterns* J:

P ::=		**processes**	J ::=		**join patterns**
	0	(nil)		$x(\widetilde{u} \,@\, \widetilde{v})$	(input)
$\mid$	$\overline{x}\,\widetilde{u}$	(output)	$\mid$	$J \,\&\, J$	(join)
$\mid$	$\sum_{i \in I} J_i \triangleright P_i$	(orchestrator)			
$\mid$	$(x \,@\, y)P$	(new)			
$\mid$	$P \mid P$	(parallel)			
$\mid$	$!P$	(replication)			

In the rest of the paper, we write $\prod_{i \in 1..n} P_i$ for $P_1 \mid \cdots \mid P_n$ and $J_1 \triangleright P_1 + \cdots + J_n \triangleright P_n$ for $\sum_{i \in 1..n} J_i \triangleright P_i$. We also write $(x)P$ for $(x \,@\, x)P$ and $x(u)$ for $x(u \,@\, u)$.

Free and bound names are standard: x is *bound* in $(x \,@\, y)P$ and $\widetilde{u}$ is bound in $x(\widetilde{u} \,@\, \widetilde{v})$; names are *free* when they occur non-bound. Write $\mathtt{bn}(P)$ and $\mathtt{bn}(J)$ for the bound names of P and J, respectively; similarly write $\mathtt{fn}(P)$ and $\mathtt{fn}(J)$ for the free names. For example, $\mathtt{fn}(x(u \,@\, u) \,\&\, y(v \,@\, u)) = \mathtt{fn}(x(v \,@\, u) \,\&\, y(u \,@\, u)) = \{x, y\}$. The scope of the name x in $(x \,@\, y)P$ is y and the process P; the scope of a name bound by J in $J \triangleright P$ is J and P. The name x in $\overline{x}\,\widetilde{u}$ and in $x(\widetilde{u} \,@\, \widetilde{v})$ is called *subject*; $\mathtt{sn}(J)$ collects all the subjects of J. The names $\bigcup_{i \in I} \widetilde{u}_i$ in $\&_{i \in I} x_i(\widetilde{u}_i \,@\, \widetilde{v}_i)$ are called *defined names*.

Processes define the computational entities of the calculus. Most of the operators are standard from the pi calculus [11], except new $(x \,@\, y)P$, orchestrator $\sum_{i \in I} J_i \triangleright P_i$, and input $x(\widetilde{u} \,@\, \widetilde{v})$. The process $(x \,@\, y)P$ creates a channel x at the same location of y. The process $(x \,@\, x)P$ creates a channel x at a fresh location. The term $x \,@\, y$ in new and input is called *co-location pair*. Orchestrators are reminiscent of join calculus definitions [6] and pi calculus input guarded choices. A branch $J_i \triangleright P_i$ is chosen provided a pattern of outputs that matches with J_i is present. In this case the continuation P_i is run and all the other alternatives are discarded. A pattern of outputs $\overline{x_1}\,\widetilde{u}_1 \mid \cdots \mid \overline{x_n}\,\widetilde{u}_n$ matches with $x_1(\widetilde{v}_1 \,@\, \widetilde{w}_1) \,\&\, \cdots \,\&\, x_n(\widetilde{v}_n \,@\, \widetilde{w}_n)$ provided $\widetilde{u}_i$ and $\widetilde{v}_i$ have the same length and the location constraints in $\widetilde{w}_i$ are satisfied. For example, $\overline{x}\,u \mid \overline{z}\,u'$ matches with $x(v) \,\&\, z(v' \,@\, v)$ if u and u' are two co-located channels.

Join patterns J satisfy the following *well-formedness constraints*:

1. *defined names of J are pairwise different;*
2. *(left-constraining) if $J = \&_{i \in 1..n} x_i(\widetilde{u}_i \,@\, \widetilde{v}_i)$ then $(\widetilde{u}_1 \cdots \widetilde{u}_n \,@\, \widetilde{v}_1 \cdots \widetilde{v}_n)$ is such that, for every decomposition $(\widetilde{u}' \,@\, \widetilde{v}')(u \,@\, v)(\widetilde{u}'' \,@\, \widetilde{v}'')$ of it, we have $v \notin \widetilde{u}''$. With an abuse of terminology, a co-location sequence that satisfies this property is said left-constraining.*

Remark 1. Left-constraining makes $\&$ not commutative. For example the pattern $J = x(u @ u) \& y(v @ u)$ is well-formed while $J' = y(v @ u) \& x(u @ u)$ is not. Left-constraining makes join patterns parsable from left to right; on the contrary, in $y(v @ u) \& x(u @ u)$, to find the binder for the occurrence of u in $y(v @ u)$ one has to read the whole join pattern. Left-constraining also simplifies some technical discussions later in the paper.

Remark 2. The well-formedness condition on join patterns does not enforce their linearity with respect to subject names. For example, the pattern $x(u) \& x(v)$ is well-formed. This linearity constraint is not easy to formalize because, in our calculus, received names may be used as subjects of inputs in the continuations – *input capability*. Removing the feature of input capability, the linearity constraint may be defined as in join calculus [6].

2.2 Co-location Relation

Process reduction is possible if the co-location constraints specified in the join pattern are fulfilled. This fulfillment is defined in terms of the *co-location relation*.

Definition 1. *Let* $\widetilde{x} @ \widetilde{y} \vdash u \frown v$*, called the* co-location relation*, be the equivalence relation on names that is induced by the following rules:*

$$\text{(BASE)}\quad (\widetilde{x} @ \widetilde{y})(u @ v) \vdash u \frown v \qquad\qquad \text{(LIFT)}\quad \frac{\widetilde{x} @ \widetilde{y} \vdash u \frown v \quad u, v \neq z}{(\widetilde{x} @ \widetilde{y})(z @ z') \vdash u \frown v}$$

For example $(x @ y)(z @ y) \vdash x \frown z$ by transitivity. A less evident entailment is $(x @ y)(z @ y)(y @ u) \vdash x \frown z$. This is due to $(x @ y)(z @ y) \vdash x \frown z$ and to the (LIFT) rule because $x, z \neq y$. We write $\widetilde{x} @ \widetilde{y} \vdash u_1 \cdots u_n \frown v_1 \cdots v_n$ if $\widetilde{x} @ \widetilde{y} \vdash u_i \frown v_i$ for every i.

The co-location relation induces a partition on names that is left informal in this contribution. For instance $(x @ y)(z @ y)(y @ u)$ gives the partition $\{x, z\}, \{y, u\}$. There are permutations of co-location sequences that preserve the induced partition. A relevant one is the following.

Proposition 1. *If* $x \neq x'$*,* $x \neq y'$*, and* $x' \neq y$ *then* $\Lambda(x @ y)(x' @ y') \vdash u \frown v$ *implies* $\Lambda(x' @ y')(x @ y) \vdash u \frown v$*.*

Proof. For brevity we only examine four cases, and we assume $u \neq v$.

($u, v \neq x, x'$) From $\Lambda(x @ y)(x' @ y') \vdash u \frown v$ and the hypotheses we derive $\Lambda \vdash u \frown v$. From this and the hypotheses we conclude $\Lambda(x' @ y')(x @ y) \vdash u \frown v$ by (LIFT).

($u = x, u, v \neq x'$) From $\Lambda(u @ y)(x' @ y') \vdash u \frown v$ we derive $\Lambda(u @ y) \vdash u \frown v$. There are two sub-cases:

- ($v = y$) We conclude $\Lambda(x' @ y')(u @ v) \vdash u \frown v$ by (BASE).
- ($v \neq y$) Since $u \frown y$ we must have derived $u \frown v$ by transitivity from $\Lambda \vdash v \frown y$. From $\Lambda \vdash v \frown y$ and the hypotheses $v, y \neq x', u$ we get $\Lambda(x' @ y')(u @ y) \vdash v \frown y$. From this and $\Lambda(x' @ y')(u @ y) \vdash u \frown y$ by transitivity we obtain $\Lambda(x' @ y')(u @ y) \vdash u \frown v$.

($u = x', u, v \neq x$) We have $\Lambda(x \,@\, y)(u \,@\, y') \vdash u \frown v$. There are two sub-cases:

- **($v = y'$)** We have $\Lambda(u \,@\, v) \vdash u \frown v$. From this and the hypotheses $u, v \neq x$ we conclude $\Lambda(u \,@\, v)(x \,@\, y) \vdash u \frown v$.
- **($v \neq y'$)** Since $u \frown y'$ we must have derived $u \frown v$ by transitivity from $\Lambda(x \,@\, y) \vdash v \frown y'$. From this and the hypotheses $v, y' \neq x$ we get $\Lambda \vdash v \frown y'$. From this and the hypotheses $v, y' \neq u, x$ we get $\Lambda(u \,@\, y')(x \,@\, y) \vdash v \frown y'$. Using similar arguments we derive $\Lambda(u \,@\, y')(x \,@\, y) \vdash u \frown y'$ and by transitivity we conclude $\Lambda(u \,@\, y')(x \,@\, y) \vdash u \frown v$.

($u = x, v = x'$) We have $\Lambda(u \,@\, y)(v \,@\, y') \vdash u \frown v$. From the hypotheses we have $u, v \neq y'$ and $u, v \neq y$. We must have concluded $u \frown v$ by transitivity from $y \frown y'$. From this and $y, y' \neq v, u$ we get $\Lambda(v \,@\, y')(u \,@\, y) \vdash y \frown y'$. From this and $\Lambda(v \,@\, y')(u \,@\, y) \vdash u \frown v$ we conclude $\Lambda(v \,@\, y')(u \,@\, y) \vdash u \frown v$. □

It is possible to establish a relation between the partition induced by a co-location sequence and the one obtained when bound names in a suffix of the same sequence are substituted.

Proposition 2. *Let ρ be a substitution such that $\Lambda \vdash \widetilde{x}\rho \frown \widetilde{y}\rho$. Then $\Lambda(\widetilde{x} \,@\, \widetilde{y}) \vdash u \frown v$ implies $\Lambda \vdash u\rho \frown v\rho$.*

Proof. By induction on the proof of $\Lambda(\tilde{x} \,@\, \tilde{y}) \vdash u \frown v$. The base case is straightforward. The inductive case is when the last rule is an instance of (LIFT). Let $(\widetilde{x} \,@\, \widetilde{y}) = (\widetilde{x}'' \,@\, \widetilde{y}'')(x' \,@\, y')$ and let $\Lambda(\widetilde{x} \,@\, \widetilde{y}) \vdash u \frown v$ be demonstrated by (LIFT) with premises $\Lambda(\widetilde{x}'' \,@\, \widetilde{y}'') \vdash u \frown v$ and $u, v \neq x'$. We have that $\Lambda \vdash \widetilde{x}\rho \frown \widetilde{y}\rho$ implies $\Lambda \vdash \widetilde{x}''\rho \frown \widetilde{y}''\rho$. Henceforth, by inductive hypothesis, we conclude $\Lambda \vdash u\rho \frown v\rho$. □

2.3 Operational Semantics

The operational semantic is defined by means of a structural congruence that equates all processes that have essentially the same structure and that are never distinguished.

Definition 2. *Structural congruence $\equiv$ is the smallest equivalence relation that satisfies the following axioms and is closed with respect to contexts and alpha-renaming:*

$$P \mid \mathtt{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad !P \equiv P \mid !P$$
$$(x \,@\, y)\mathtt{0} \equiv \mathtt{0} \qquad (x \,@\, y)(P \mid Q) \equiv P \mid (x \,@\, y)Q \quad \textit{if } x \notin \mathtt{fn}(P)$$
$$(x \,@\, y)(x' \,@\, y')P \equiv (x' \,@\, y')(x \,@\, y)P \quad \textit{if } x \neq x',\, x \neq y',\, \textit{and } y \neq x'$$

Notice that the last congruence axiom is strictly related to Proposition 1.

Definition 3. *The reduction relation $\rightarrow$ is the least relation satisfying the rule*

$$\frac{M = \prod_{j \in 1..n} \overline{x_j}\,\widetilde{u}_j \quad J_k = \&_{j \in 1..n} x_j(\widetilde{u}_j \,@\, \widetilde{v}_j) \quad k \in I \qquad \mathtt{dom}(\rho) = \bigcup_{j \in I} \widetilde{u}_j \quad \left(\widetilde{z} \,@\, \widetilde{y} \vdash \widetilde{u}_j\rho \frown \widetilde{v}_j\rho\right)^{j \in 1..n}}{(\widetilde{z} \,@\, \widetilde{y})\Big(M\rho \mid \sum_{i \in I} J_i \triangleright P_i \mid R\Big) \rightarrow (\widetilde{z} \,@\, \widetilde{y})\Big(P_k\rho \mid R\Big)}$$

and closed under $\equiv$, $_ \mid _$ and $(z \,@\, z')_$.

The last premise of the reduction rule requires that the interacting processes are underneath a sequence of news that is long enough. For example, the process $P = \overline{x}\, u \mid x(u \,@\, v) \triangleright Q$ is inactive. On the contrary $(u \,@\, v)P$ reduces to Q. We refer to the introduction for few sample processes in our calculus.

The semantics is completed by the notion of *barbed congruence* [12]. According to this notion, two processes are considered equivalent if their reductions match and they are indistinguishable under global observations and under any context.

Definition 4. *The name x is a* barb *of P, written $P \downarrow x$, when*

$$\begin{array}{ll} \overline{x}\,\widetilde{u} \downarrow x & \\ (z \,@\, y)P \downarrow x & \text{if } P \downarrow x \text{ and } x \neq z \\ !P \downarrow x & \text{if } P \downarrow x \\ P \mid Q \downarrow x & \text{if } P \downarrow x \text{ or } Q \downarrow x \end{array}$$

Write $\Rightarrow$ for $\rightarrow^*$ and $\Downarrow$ for $\Rightarrow\downarrow$.

A barbed bisimulation *is a symmetric relation ϕ such that whenever $P \;\phi\; Q$ then (1) $P \downarrow x$ implies $Q \Downarrow x$, and (2) $P \rightarrow P'$ implies $Q \Rightarrow Q'$ and $P' \;\phi\; Q'$. The largest barbed bisimulation is noted $\dot{\approx}$.*

Let $C[\,]$ be the set of contexts generated by the grammar:

$$C[\,] \;::=\; [\cdot] \;\mid\; \sum_{i \in I} J_i \triangleright C[\,] \;\mid\; (x \,@\, y)C[\,] \;\mid\; P|C[\,] \;\mid\; C[\,]|P \;\mid\; !C[\,]$$

The barbed congruence *is the largest symmetric relation $\approx$ such that whenever $P \approx Q$ then, for all contexts $C[\,]$, $C[P] \;\dot{\approx}\; C[Q]$.*

3 On the Expressivity of Orchestrators

Orchestrators constitute a basic mechanism that may express standard workflow patterns in Web services as well as common synchronization constructs in programming languages. A few paradigmatic encodings of patterns are described below.

Client/supplier/bank interaction. The first example describes a $\mathit{Supplier}$ that waits for requests from clients. Upon receiving a buy request, the supplier asks the client about his financial availability. The client must reassure the supplier by letting his financial institution (a bank) vouch directly for him. In the meantime, the supplier forwards the client's request to the appropriate manufacturer, which will proceed with the delivery as soon as he receives a confirmation from the bank. We write $\overline{x}\,[\widetilde{u}]$ instead of $\overline{x}\,\widetilde{u}$:

$$\begin{array}{rl} \mathit{Supplier} \stackrel{\text{def}}{=} & \mathit{buy}(\mathit{item}, x) \triangleright (\mathit{voucher} \,@\, \mathit{item})(\\ & \qquad \overline{x}\,[\mathit{voucher}, \mathit{amount}] \\ & \qquad \mid \mathit{voucher}(u) \,\&\, \mathit{item}(v) \triangleright \overline{\mathit{deliver}}\,[u, v] \mid \overline{\mathit{record}}\,[u, v]) \end{array}$$

We note that several clients may compete for the same item. In this case, delivery occurs only when the payment for the item is available. We also observe that the channel $\mathit{voucher}$ is co-located with item. Henceforth, the orchestrator $\mathit{voucher}(u) \,\&\, \mathit{item}() \triangleright \overline{\mathit{deliver}}\,[u, v] \mid \overline{\mathit{record}}\,[u, v]$ will coordinate two channel managers at a same location.

Synchronizing merge. *Synchronizing merge* is one of the advanced synchronization patterns in [14]. According to this pattern, an activity A may trigger one or two concurrent activities B and C. These activities B and C signal their completion by sending messages to a fourth activity D. Synchronization occurs only if both B and C have been triggered. We assume that the choice of A of triggering one between B and C or both is manifested to D by emitting a signal over one or not, respectively. This signal is similar to the so-called "false tokens" in those workflow engines that support this synchronization pattern:

$$\mathit{SynchMerge} \stackrel{\text{def}}{=} b(v) \mathbin{\&} \mathit{one}() \triangleright \overline{d}\,[v] + c(v) \mathbin{\&} \mathit{one}() \triangleright \overline{d}\,[v] + b(v) \mathbin{\&} c(v') \triangleright \overline{d}\,[v, v']$$

Dynamic load balancing. Our last example models a load balancing mechanism for Web services. We assume the existence of two message queues: job, where requests for services are posted, $ready$ where services make themselves available for processing one or more requests:

$$\mathit{LoadBal} \stackrel{\text{def}}{=} \ldots \mathit{ready}(w) \mathbin{\&} \mathit{job}(u) \triangleright \overline{w}\,[u]$$

This is a typical load balancing mechanism that can handle multiple requests concurrently, or may distribute the computational load among different servers, possibly depending on request's priority. In addition, in our language it is possible to change the load dynamically. For instance, a supplier could run the code

$$\mathit{job}(u) \mathbin{\&} \mathit{job}(v) \triangleright \overline{w}\,[u, u']$$

that changes the policy by processing two jobs at a same time. This small piece of code – a smooth orchestrator – may migrate to the location of the load balance process in order to update the current policy.

4 The Smoothness Restriction

A distributed prototype of the calculus in Section 2 may be designed with difficulties. Let us commit to a standard abstract machine of several distributed implementations of process calculi [15, 6, 4]. Such a machine consists of processors running at different locations with channels that are uniquely located to processors. Outputs are always delivered to the processor of the corresponding subject where they may be consumed. In this machine, the process

$$x(u \,@\, u, v \,@\, v) \;\triangleright\; \big(u(w) \mathbin{\&} v(w') \triangleright P\big)$$

dynamically creates an orchestrator on the received channels u and v. There are at least two problematic issues as far as distribution is concerned. Let z and y be the names respectively replacing u and v at run-time:

1. the orchestrator $z(w) \mathbin{\&} y(w') \triangleright P$ is consuming inputs on channels z and y that may be not co-located. This means that the processors running such channel(-manager)s must compete with the processor running $z(w) \mathbin{\&} y(w') \triangleright P$ for consuming output. This is a classical global consensus problem.

2. if the channels z and y were co-located, the global consensus problem could be solved by migrating the orchestrator $z(w) \,\&\, y(w') \triangleright P$ to the right processor. However, this migration is expensive, because P may be large and could require a large closure.

Due to these problems, it is preferable to restrain our study to a sub-calculus of that in Section 2, which is more amenable to distributed implementations. The restrictions we consider are the following two:

1. every orchestrator is *smooth*, namely it has the shape $\sum_{i\in I} J_i \triangleright \overline{z_i}\,\widetilde{u}_i$ where $\widetilde{u}_i$ is the sequence of bound names in J_i in the same order as they appear in J_i and $\bigcup_{i\in I} \mathtt{sn}(J_i)$ are all co-located;
2. we admit processes $z(\widetilde{u} \,@\, \widetilde{v}) \triangleright P$, namely generic continuations are restricted to simple inputs.

The formalization of the co-location restriction of joins in smooth orchestrators is defined by means of the type system in Table 1. Let ε denote the empty co-location sequence.

Table 1. Co-location checks for the full calculus

(NIL) $\Lambda \vdash 0$ (OUTPUT) $\Lambda \vdash \overline{x}\,\widetilde{u}$

(ORCH)
$$\frac{(\vdash J_i :: \Lambda_i \quad \Lambda\Lambda_i \vdash P_i)^{i\in I} \quad (\Lambda \vdash x \frown y)^{x\in \mathtt{sn}(J_i), y \in \mathtt{sn}(J_j)}}{\Lambda \vdash \sum_{i\in I} J_i \triangleright P_i}$$

(NEW)
$$\frac{\Lambda(x \,@\, y) \vdash P}{\Lambda \vdash (x \,@\, y)P}$$

(PAR)
$$\frac{\Lambda \vdash P \quad \Lambda \vdash Q}{\Lambda \vdash P \mid Q}$$

(BANG)
$$\frac{\Lambda \vdash P}{\Lambda \vdash !P}$$

(INPUT)
$$\vdash x(\widetilde{u} \,@\, \widetilde{v}) :: \widetilde{u} \,@\, \widetilde{v}$$

(JOIN)
$$\frac{\vdash J :: \Lambda' \quad \vdash J' :: \Lambda''}{\vdash J \,\&\, J' :: \Lambda'\Lambda''}$$

Definition 5. *A process P is* distributable *if $\varepsilon \vdash P$.*

We defer the analysis of the distributed implementation of smooth orchestrators to Section 5. The rest of the section is devoted to the correctness of the co-location system and the encoding of the calculus in Section 2 into the sub-calculus with smooth orchestrators. We begin with a couple of technical statements.

Lemma 1. *1. Let Λ and Λ' be such that, for every $x, y \in \mathtt{fn}(P)$, if $\Lambda \vdash x \frown y$ then $\Lambda' \vdash x \frown y$. Then $\Lambda \vdash P$ implies $\Lambda' \vdash P$.*
2. Let x' be fresh with respect to names in Λ and in $\mathtt{fn}(P)$. Then $\Lambda(x \,@\, y) \vdash P$ implies $\Lambda(x' \,@\, y\{^{x'}/_{x}\}) \vdash P\{^{x'}/_{x}\}$.

Proof. We prove item 2; the first is simpler. The argument is by induction on the proof of $\Lambda(x \,@\, y) \vdash P$ and we discuss the case when the last rule is an instance of (NEW); the others are similar or straightforward.

In this case $P = (u \mathbin{@} v)P'$ and $\Lambda(x \mathbin{@} y) \vdash (u \mathbin{@} v)P'$. By (NEW) one reduces to $\Lambda(x \mathbin{@} y)(u \mathbin{@} v) \vdash P'$. There are two sub-cases (a) $u \neq x, y$ and $v \neq x$ and (b) the others. In (a), by Proposition 1 and item 1, we also have $\Lambda(u \mathbin{@} v)(x \mathbin{@} y) \vdash P'$. Then, by inductive hypotheses, it is possible to obtain $\Lambda(u \mathbin{@} v)(x' \mathbin{@} y\{^{x'}/x\}) \vdash P'\{^{x'}/x\}$. Using again item 1 we derive $\Lambda(x' \mathbin{@} y\{^{x'}/x\})(u \mathbin{@} v) \vdash P'\{^{x'}/x\}$ and we conclude by (NEW). The sub-case (b) is proved as follows. Let $\Lambda(x \mathbin{@} y)(u \mathbin{@} v) \vdash P'$. The clashes of u with x or y may be removed by inductive hypothesis. Therefore, the problematic case is $\Lambda(x \mathbin{@} y)(u \mathbin{@} x) \vdash P'$.

If $x \neq y$ then consider the co-location sequence $\Lambda(u \mathbin{@} y)(x \mathbin{@} y)$. It is easy to prove that $\Lambda(x \mathbin{@} y)(u \mathbin{@} x) \vdash u' \frown v'$ implies $\Lambda(u \mathbin{@} y)(x \mathbin{@} y) \vdash u' \frown v'$. Therefore, by item 1 we may derive $\Lambda(u \mathbin{@} y)(x \mathbin{@} y) \vdash P$ and, by inductive hypothesis, we derive $\Lambda(u \mathbin{@} y)(x' \mathbin{@} y\{^{x'}/x\}) \vdash P\{^{x'}/x\}$. With a same argument it is possible to obtain $\Lambda(x' \mathbin{@} y\{^{x'}/x\})(u \mathbin{@} x') \vdash P\{^{x'}/x\}$ and by (NEW) we conclude $\Lambda(x' \mathbin{@} y\{^{x'}/x\}) \vdash (u \mathbin{@} x')P\{^{x'}/x\}$.

If $x = y$ then we consider the co-location sequence $\Lambda(u \mathbin{@} u)(x \mathbin{@} u)$. The proof may be completed as in the previous sub-case. □

It is worth to notice that Lemma 1 entails the weakening statement: if $\Lambda \vdash P$ and x is fresh then $\Lambda(x \mathbin{@} y) \vdash P$.

Lemma 2 (substitution). *Let ρ be a substitution such that $\mathtt{dom}(\rho) = \widetilde{x}$. If $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash P$ and $\Lambda \vdash \widetilde{x}\rho \frown \widetilde{y}\rho$ then $\Lambda \vdash P\rho$.*

Proof. The argument is by induction on the proof of $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash P$. The interesting cases are when the last rule is an instance of (NEW) and of (ORCH).

(NEW). Let $P = (u \mathbin{@} v)P'$ and $u \neq v$ (the case $u = v$ is similar). By (NEW) we are reduced to

$$\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v) \vdash P' \tag{1}$$

There are a number of sub-cases:

- $u \notin \widetilde{x}\widetilde{y}$ and $v \notin \widetilde{x}$. Then by Proposition 1 the contexts $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v)$ and $\Lambda(u \mathbin{@} v)(\widetilde{x} \mathbin{@} \widetilde{y})$ are equivalent and by item 1 we have $\Lambda(u \mathbin{@} v)(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash P'$. By applying the inductive hypothesis and (NEW) we obtain $\Lambda \vdash (u \mathbin{@} v)(P\rho)$ that leads to $\Lambda \vdash ((u \mathbin{@} v)P)\rho$ since $u, v \notin \widetilde{x}$.
- $u \notin \widetilde{x}\widetilde{y}$ and $v \in \widetilde{x}$. We discuss two possibilities. If $(v \mathbin{@} w) \in (\widetilde{x} \mathbin{@} \widetilde{y})$ and $w \notin \widetilde{x}$ the contexts $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v)$ and $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} w)$ are equivalent and we can apply the same arguments as for the previous case. If $(v \mathbin{@} v) \in (\widetilde{x} \mathbin{@} \widetilde{y})$ consider the context $\Lambda(u \mathbin{@} v\rho)(\widetilde{x}' \mathbin{@} \widetilde{y}')$ where $(\widetilde{x}' \mathbin{@} \widetilde{y}')$ is obtained by substituting $(v \mathbin{@} v)$ with $(v \mathbin{@} u)$ in $(\widetilde{x} \mathbin{@} \widetilde{y})$. Note that $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v) \vdash u' \frown v'$ implies $\Lambda(u \mathbin{@} v\rho)(\widetilde{x}' \mathbin{@} \widetilde{y}') \vdash u' \frown v'$ so we can apply item 1 followed by the inductive hypothesis and obtain $\Lambda(u \mathbin{@} v\rho) \vdash P'\rho$. From this we derive $\Lambda \vdash (u \mathbin{@} v\rho)P'\rho$, which is equivalent to $\Lambda \vdash ((u \mathbin{@} v)P')\rho$.
- $u \in \widetilde{x}\widetilde{y}$. By Lemma 1(2) we reduce to $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u' \mathbin{@} v) \vdash P'\{^{u'}/u\}$, where u' is fresh, therefore $u' \notin \widetilde{x}\widetilde{y}$. In the same way as in the first sub-case, we obtain $\Lambda \vdash ((u' \mathbin{@} v)P'\{^{u'}/u\})\rho$ and we conclude because, by definition of substitution, $((u' \mathbin{@} v)P'\{^{u'}/u\})\rho = ((u \mathbin{@} v)P')\rho$ when $u \in \widetilde{x}\widetilde{y}$.

(ORCH). We discuss the case when $P = J \triangleright P'$. The general case is similar. By (ORCH), and letting $\vdash J :: (\widetilde{z} \,@\, \widetilde{w})$ we are reduced to

$$\Lambda(\widetilde{x} \,@\, \widetilde{y})(\widetilde{z} \,@\, \widetilde{w}) \vdash P' \tag{2}$$

and

$$(\Lambda(\widetilde{x} \,@\, \widetilde{y}) \vdash x' \frown y')^{x',y' \in \mathtt{sn}(J)} \tag{3}$$

We may apply Proposition 2 to (3) and obtain

$$(\Lambda \vdash x'\rho \frown y'\rho)^{x',y' \in \mathtt{sn}(J)} \tag{4}$$

From (2), with a similar argument as in (NEW), we may derive $\Lambda(\widetilde{z} \,@\, \widetilde{w}\rho) \vdash P'\rho$ or similar judgments renaming names in $\widetilde{z}$ when they clash with $\widetilde{x}\widetilde{y}$ (we omit these last cases). By applying (ORCH) to this judgment, to (4) and to $\vdash J\rho :: (\widetilde{z} \,@\, \widetilde{w}\rho)$ we therefore derive $\Lambda \vdash J\rho \triangleright P'\rho$. We conclude by observing that $J\rho \triangleright P'\rho = (J \triangleright P')\rho$. □

A brief discussion about the substitution lemma follows. Consider

$$(a \,@\, a)(u \,@\, u)(v \,@\, u) \vdash u \,\&\, v \triangleright 0$$

and the substitution $\{a/v\}$. Note that $(a \,@\, a)(u \,@\, u) \not\vdash v\{a/v\} \frown u\{a/v\}$. Indeed, if we were allowed to apply $\{a/v\}$ to the above judgment, we would obtain

$$(a \,@\, a)(u \,@\, u) \not\vdash u \,\&\, a \triangleright 0$$

Actually, the process $u \,\&\, v \triangleright 0$ is well-typed in a context that co-locates u and v. While this is the case for $(a \,@\, a)(u \,@\, u)(v \,@\, u)$, it is not the case for $(a \,@\, a)(u \,@\, u)$. The condition $\Lambda \vdash \widetilde{x}\rho \frown \widetilde{y}\rho$ in the substitution establishes that co-located names remain co-located after having been substituted. Therefore, if we insist in replacing v with a, we must also map u to a. In this case the substitution lemma may be applied and we obtain:

$$(a \,@\, a) \vdash a \,\&\, a \triangleright 0$$

Theorem 1 (subject reduction). *If* $(\widetilde{x} \,@\, \widetilde{y}) \vdash P$ *and* $(\widetilde{x} \,@\, \widetilde{y})P \to (\widetilde{x} \,@\, \widetilde{y})Q$ *then* $(\widetilde{x} \,@\, \widetilde{y}) \vdash Q$*. In particular, if* P *is distributable and* $P \to Q$ *then* Q *is distributable as well.*

Proof. It is sufficient to show that well-typedness is preserved by any structural congruence rule (in both directions) and by the reduction rule. We omit the easy cases.

- Let $\Lambda \vdash (x \,@\, y)(x' \,@\, y')P$ with $x \neq y'$, $x \neq x'$, and $y \neq x'$. By (NEW): $\Lambda(x \,@\, y)(x' \,@\, y') \vdash P$. By Lemma 1(1): $\Lambda(x' \,@\, y')(x \,@\, y) \vdash P$. We conclude by (NEW).
- Let $\Lambda \vdash (x \,@\, y)(P \mid Q)$ and $x \notin \mathtt{fn}(P)$. It is sufficient to show that if $u, v \neq x$ then $\Lambda \vdash u \frown v$ iff $\Lambda(x \,@\, y) \vdash u \frown v$. This follows by the rule (LIFT) of the co-location relation.

- Let $\Lambda \vdash (M\rho \mid \sum_{i\in I} J_i \triangleright P_i \mid R)$ and let $(\Lambda)(M\rho \mid \sum_{i\in I} J_i \triangleright P_i \mid R) \rightarrow (\Lambda)(P_k\rho \mid R)$. By the hypotheses of the reduction rule: $M = \prod_{j=1..n} \overline{x_j}\, u_j$, $J_k = \&_{j\in 1..n} x_j(\widetilde{u}_j \mathbin{@} \widetilde{v}_j)$, $\mathtt{dom}(\rho) = \bigcup_{j=1..n} \widetilde{u}_j$ and $\Lambda \vdash \widetilde{u}_j\rho {}^\frown \widetilde{v}_j\rho$ for all $j \in 1..n$. The type system yields $\vdash J_k :: (\tilde{u}_j \mathbin{@} \tilde{v}_j)^{j\in 1..n}$. Therefore, by the Substitution Lemma applied to $\Lambda(\tilde{u}_j \mathbin{@} \tilde{v}_j)^{j\in 1..n} \vdash P_k$, we obtain $\Lambda \vdash P_k\rho$. From this we conclude $\Lambda \vdash P_k\rho \mid R$. □

The calculus with distributable orchestrators may be encoded into the calculus with smooth ones. We first define an encoding that decouples complex continuations from join patterns.

Definition 6. *The encoding $[\![\cdot]\!]$ is defined on processes in Section 2. The function $[\![\cdot]\!]$ is an homomorphism except for orchestrators. In the definition below we assume that, for every j, $z_j \notin \bigcup_{i\in I}(\mathtt{fn}(J_i) \cup \mathtt{bn}(J_i))$ and the tuple $\widetilde{u}_j$ is exactly the sequence of defined names in J_j:*

$$[\![\sum_{i\in I} J_i \triangleright P_i]\!] \;=\; (z_i{}^{i\in I})\Big(\sum_{i\in I} J_i \triangleright \overline{z_i}\,\widetilde{u}_i \;\mid\; z_i(\widetilde{u}_i) \triangleright [\![P_i]\!]\Big)$$

It is folklore to demonstrate the correctness of the encoding $[\![\cdot]\!]$, namely $P \approx Q$ if and only if $[\![P]\!] \approx [\![Q]\!]$. This is an immediate consequence of the following statement, that in turn uses a generalization of the pi calculus law $x(\widetilde{u}).P \approx (z)(x(\widetilde{u}).\overline{z}\,\widetilde{u} \mid z(\widetilde{u}).P)$.

Proposition 3. *For every P, $P \approx [\![P]\!]$.*

Of course, if P is a generic process with orchestrators then join patterns in $[\![P]\!]$ may have subjects that are not co-located. It is possible to avoid such problematic cases by restricting the domain of $[\![\cdot]\!]$ to distributable processes.

Proposition 4. *If P is distributable then $[\![P]\!]$ is a process with smooth orchestrators.*

5 The Implementation of Smooth Orchestrators

Smooth orchestrators are small pieces of code that may migrate over the network for reaching the location where they execute. Unlike mobile agents, they exhibit a simple, finite behavior and they require a limited-size message to migrate. Consider a single branch smooth orchestrator:

$$x_1(\widetilde{u}_1 \mathbin{@} \widetilde{v}_1) \,\&\, \cdots \,\&\, x_n(\widetilde{u}_n \mathbin{@} \widetilde{v}_n) \triangleright \overline{z}\,\widetilde{u}_1 \cdots \widetilde{u}_n$$

It may be encoded as a vector of $n+1$ names – the subjects $x_1, \cdots, x_n$ plus the destination channel z – and a vector of $k_1 + \cdots + k_n$ values, where k_i is the length of the tuple $\widetilde{u}_i$. Each value can be either an integer or a (free) name and it encodes a co-location constraint: (1) the integer value j at position h indicates that the j-th and h-th bound names must be co-located; (2) the constant c at position h indicates that the h-th bound name must be co-located with c. An orchestrator of m branches is encoded by a vector of length m whose elements are pairs of vectors of the above shape.

The destination of this vector is driven by the location of the subjects (remember that the subjects are co-located). When this vector arrives at destination, it triggers an appropriate process that monitors the states of the message queues of the subjects. We discuss the implementation of smooth orchestrators of increasing complexity, starting from the automata-based technique for implementing join patterns in the join calculus, and gradually extending the technique to include the new features. Initially we omit the discussion of nonlinear patterns and orchestrators with multiple branches.

In the join calculus a join definition is compiled as a finite state automaton that keeps track of the status of the message queues associated with the corresponding channels [9]. Formally, let $x_1(\widetilde{u}_1) \,\&\, \cdots \,\&\, x_n(\widetilde{u}_n) \triangleright \overline{z}\,\widetilde{u}_i$ be the definition, which is also a smooth orchestrator. The associated automaton is

$$M = (\wp(\{x_1, \dots, x_n\}), \{+x_1, -x_1, \dots, +x_n, -x_n\}, \delta, \emptyset, \{x_1, \dots, x_n\}) \qquad (5)$$

where

$$\delta(q, +x_i) = q \cup \{x_i\} \qquad \delta(q, -x_i) = q \setminus \{x_i\}$$

The automaton reacts to symbols of the form $+x$, meaning "the message queue of the channel x is not empty" and $-x$, meaning "the message queue of the channel x is empty". Every time a message queue changes (either because a new message arrives, or because a message is removed) it notifies all the automata associated with it. When the joined channels are all fresh (and join definitions cannot be extended at runtime, like in the join calculus) there will be a unique automaton for handling the whole definition. In our case channel orchestrations may be added and/or removed at runtime, thus making the set of automata associated with them change over time. The consequent competitions for messages in shared channel queues are solved without difficulties because the automata are all co-located.

This mechanism may be easily extended with co-location constraints when the scope of such constraints is limited to the message itself. This is the case in $x(u \,@\, u, v \,@\, u) \,\&\, y(w \,@\, w) \triangleright P$. To model this extension the alphabet of the automata is patched by admitting symbols of the form $+x(\widetilde{u} \,@\, \widetilde{v})$ and $-x(\widetilde{u} \,@\, \widetilde{v})$ instead of $+x$ and $-x$. When a new message $\overline{x}\,\widetilde{a}$ is available, each automaton associated with x makes a $+x(\widetilde{u} \,@\, \widetilde{v})$ transition only if the co-location constraints are satisfied. When a message $\overline{x}\,\widetilde{a}$ is removed from the x-queue, every automaton that has been affected by the message checks whether the queue contains another message satisfying its co-location constraints or not. In case there is no such message, the state of the automaton is adequately reset in accordance with the new state of the queue.

When different joined channels have co-location dependencies, the constraints to be verified may involve names that have been bound during previous transitions. For example, take $x(u \,@\, u) \,\&\, y(v \,@\, u)$ and assume that the corresponding automaton has made a transition on a message $\overline{x}\,a$. The subsequent transition on y depends on a's location: only a message $\overline{y}\,b$ such that a and b are co-located will make the automaton move into the accepting state. Symmetrically, the automaton may start with a message $\overline{y}\,b$. In this case the automaton may progress provided a message $\overline{x}\,a$ has been enqueued with a co-located with b. In facts, we are rewriting the above pattern into $y(v \,@\, v) \,\&\, x(u \,@\, v)$, which preserves the co-location constraints (this operation is sustained by Lemma 1.1)

and is left-constraining. More precisely, in the case of $J = \&_{i \in 1..n} x_i(\widetilde{u_i} @ \widetilde{v_i})$ the automaton is defined as follows. Let $W = \bigcup_{j \in 1..n} \widetilde{u_j}\widetilde{v_j}$ and $\widetilde{w_j}$ be tuples in W. Then

$$M = (\{ \&_{i \in I} x_i(\widetilde{u_i} @ \widetilde{w_i}) \mid I \subseteq 1..n\}, \{x_i(\widetilde{u_i} @ \widetilde{w_i}) \mid i \in 1..n\}, \delta, J, \emptyset)$$

where the transition relation δ is defined by

$$\begin{array}{l} \left(\&_{i \in I} x_i(\widetilde{u_i} @ \widetilde{w_i}) \right) \& x(\widetilde{u} @ \widetilde{w}) \& \left(\&_{j \in I'} x_j(\widetilde{u_j} @ \widetilde{w_j}) \right) \\ \stackrel{x(\widetilde{u} @ \widetilde{w}')}{\longrightarrow} \quad \left(\&_{i \in I} x_i(\widetilde{u_i} @ \widetilde{w}'_i) \right) \& \left(\&_{j \in I'} x_j(\widetilde{u_j} @ \widetilde{w_j}) \right) \end{array}$$

where $(\widetilde{u_i} @ \widetilde{w_i})^{i \in I}(\widetilde{u} @ \widetilde{w})$ is equivalent (in the sense of Lemma 1.1) to the sequence $(\widetilde{u} @ \widetilde{w}')(\widetilde{u_i} @ \widetilde{w}'_i)^{i \in I}$. (This rewriting can always be accomplished with simple syntactic transformations because the join pattern is left constrained.)

The instantaneous description of an automaton is a pair (q, ρ) where q is the current state and ρ is the substitution over names that have been bound while the automaton moved from the initial state to q. The behavior of the automaton can be defined by the following transition relation between instantaneous descriptions:

$$(q, \rho) \stackrel{\overline{x}\, a_1, \ldots, a_n}{\longrightarrow} (q', \rho[u_1 \mapsto a_1] \cdots [u_n \mapsto a_n])$$

if $q' = \delta(q, x(u_1 @ v_1, \ldots, u_n @ v_n))$ and $a_i \frown v_i \rho$. Note that the behavior is not deterministic: an incoming message may spawn a new automaton at any time.

As usual this nondeterminism may be described in terms of multiple automata running simultaneously, or by means of backtracking when there is a choice. It is well known that nondeterministic automata are considerably more expensive than the deterministic ones in terms of space occupation or computational complexity. Since this complexity is unavoidable if constraints make two or more input channels depend on each other, it makes sense to look for solutions that limits the use of nondeterministic automata as much as possible. One of such solutions that we consider is the following. Given a pattern $J = \&_{i \in I} x_i(\widetilde{u_i} @ \widetilde{v_i})$ we can partition the set of x_i's so that two channels stay in the same partition only if they have co-location dependencies. Inputs that only have local co-location constraints, like $x(u @ u)$ or $y(u @ u, v @ u)$, are placed in singleton partitions. Then, a deterministic automaton can be created for handling the pattern J partition-wise. On the contrary, every partition that contains inputs with co-location dependencies will be implemented by means of a nondeterministic automaton. It turns out that this simple optimization is effective since most of the orchestrators with complex join patterns that are used in practice have very few co-location dependencies.

So far the implementation of smooth orchestrators that are not linear with respect to subjects have been purposely overlooked. The deterministic automaton 5 described above can handle nonlinear patterns following the suggestion of Maranget and Le Fessant in [9]. The basic observation is that the number of channels involved in a pattern is finite and the automata can query the associated message queues for the number of the needed messages. Because of their nature, nondeterministic automata can also handle nonlinear patterns. On the contrary, deterministic automata with co-location constraints cannot be extended in a straightforward way. Consider the pattern $x(u @ u, v @ v) \&$ $x(w @ w, z @ w)$. If a message satisfies $x(w @ w, z @ w)$, but the automaton uses it for

making a transition on $x(u \,@\, u, v \,@\, v)$, then it might be not possible to reach the accepting state. What is needed in this case is again a form of nondeterminism.

The implementation of orchestrators that consist of several branches makes use of the solution adopted in join calculus that mostly *merges* the automata for different branches into a single automaton. The idea being that if the branches involve shared inputs, the automata usually share some common structure and the resulting automaton is smaller than the sum of the automata for the branches taken separately.

References

1. Aceto, L., Bloom, B., Vaandrager, F.W.: Turning SOS rules into equations. Information and Computation **111**(1) (1994) 1–52
2. Andrews, T., et.al.: Business Process Execution Language for Web Services. Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems (2003)
3. Busi, N., Padovani, L.: A distributed implementation of mobile nets as mobile agents. In: Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005). Volume 3535 of LNCS., Springer Verlag (2005) 259–274
4. Brown, A., Laneve, C., Meredith, G.L.: PiDuce: a process calculus with native XML datatypes. In: 2nd International Workshop on Web Services and Formal Methods (WS-FM 2005). Volume 3670 of LNCS., Springer Verlag (2005) 18–34
5. Carpineti, S., Laneve, C., Milazzo, P.: The BoPi machine: a distributed machine for experimenting web services technologies. In: Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), IEEE Computer Society Press (2005) 202–211
6. Fournet, C.: The Join-Calculus: A Calculus for Distributed Mobile Programming. PhD thesis, École Polytechnique, Paris, France (1998)
7. Gardner, P., Laneve, C., Wischik, L.: Linear forwarders. In R. Amadio, D.L., ed.: CONCUR 2002. Volume 2761 of Lecture Notes in Computer Science., Springer-Verlag (2003) 415–430
8. Kavantzas, N., Olsson, G., Mischkinsky, J., Chapman, M.: Web Services Choreography Description Languages. Oracle Corporation (2003)
9. Le Fessant, F., Maranget, L.: Compiling join-patterns. In Nestmann, U., Pierce, B.C., eds.: Proceedings of High-Level Concurrent Languages ’98. Volume 16.3 of Electronic Notes in Computer Science. (1998)
10. Leymann, F.: Web Services Flow Language (wsfl 1.0). Technical report, IBM Software Group (2001)
11. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Information and Computation **100** (1992) 1–77
12. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Proceedings of ICALP ’92. Volume 623 of Lecture Notes in Computer Science., Springer-Verlag (1992) 685–695
13. Thatte, S.: XLANG: Web services for business process design. Microsoft Corporation (2001)
14. van der Aalst, W.: Workflow patterns. At `www.workflowpatterns.com` (2001)
15. Wojciechowski, P., Sewell, P.: Nomadic pict: Language and infrastructure design for mobile agents. In: Proceedings of ASA/MA ’99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents). (1999)

On the Relative Expressive Power of Asynchronous Communication Primitives*

Daniele Gorla

Dipartimento di Informatica,
Università di Roma "La Sapienza"

Abstract. In this paper, we study eight asynchronous communication primitives, arising from the combination of three features: *arity* (monadic vs polyadic data), *communication medium* (message passing vs shared dataspaces) and *pattern-matching*. Each primitive has been already used in at least one language appeared in literature; however, to uniformly reason on such primitives, we plugged them in a common framework inspired by the asynchronous π-calculus. By means of possibility/impossibility of 'reasonable' encodings, we compare every pair of primitives to obtain a hierarchy of languages based on their relative expressive power.

1 Introduction

In the last 25 years, several languages and formalisms for distributed and concurrent systems appeared in literature. Some of them (e.g., CCS [18] and the π-calculus [23]) are mostly mathematical models, mainly used to formally reason on concurrent systems; other ones (e.g., LINDA [15]) are closer to actual programming languages and are mainly focused on issues like usability and flexibility. As a consequence, the former ones are usually very essential, while the latter ones provide more sophisticated and powerful programming constructs.

Despite their differences, there are, however, some basic features that are somewhat implemented in all these languages. Roughly speaking, these features can be described as the possibility of having different *execution threads* (or *processes*) that *run concurrently* by interacting via some form of *communication*. At least at a first glance, the last feature (i.e., the inter-process communication) has yield the highest variety of proposals. These arose from the possibility of having synchronous/asynchronous primitives, monadic/polyadic data, first-order/higher-order values, dataspace-based/channel-based communication media, local/remote exchanges (whenever processes are explicitly distributed, like in [8, 11]), built-in pattern-matching mechanisms, point-to-point/broadcasting primitives, and so on. The aim of this work is to formally study some of these proposals and to organise them in a clear hierarchy, based on their expressive power. Hopefully, our results should help to understand the peculiarities of every communication primitive and, as a consequence, they could be exploited to choose the 'right' primitive when designing new languages and formalisms.

We focus on *asynchronous* communication primitives, since they are the most basilar ones. Among the remaining features mentioned above, we focus on *arity of data*,

* This work has been partially supported by Project SENSORIA, founded by EU-IST Programme, contract number 016004.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 47–62, 2006.
© Springer-Verlag Berlin Heidelberg 2006

communication medium and possibility of *pattern-matching*. The expressiveness of the omitted features has been already dealt with elsewhere [25, 11, 13]; we leave as a future work the integration of these results in our framework. Notice that we studied pattern-matching because it is nowadays becoming more and more important, especially in languages that deal with complex data like XML [1, 5, 9]. However, for the sake of simplicity, we consider here a very basic form of pattern-matching, that only checks for name equality while retrieving a datum; the formal study of more flexible and powerful mechanisms (e.g., those in [12]) is left for future work.

By combining the three features chosen, we obtain eight communication primitives that have been all already employed elsewhere, e.g. in [17, 4, 15, 8, 11, 9]. However, to uniformly reason on such primitives, we plugged them in a common framework inspired by the asynchronous π-calculus; we choose the π-calculus because nowadays it is one of the best-established workbenches for theoretical reasoning on concurrent systems. By following [26, 10, 22], we shall compare the resulting languages by means of their *relative expressive power*, i.e. we shall try to encode one in the other and study the properties of the encoding function. More precisely, we shall exploit possibility/impossibility of 'reasonable' encodings (as introduced in [22]) to compare every pair of primitives, thus obtaining a hierarchy of languages based on their relative expressive power.

Our results show that the communication paradigm underlying LINDA [15] (polyadic, dataspace-based and with pattern-matching) is at the top of the hierarchy; not incidentally, LINDA's paradigm has been used in actual programming languages [3, 14]. On the opposite extreme, we have the communication paradigm used in Ambient [8] (monadic, dataspace-based but without pattern-matching). Such a paradigm is very simple but also very poor; indeed, Ambient's expressive power mostly arises from the mobility primitives. Strictly in the middle, we find the asynchronous π-calculus (channel-based and without pattern-matching), in its monadic and polyadic version. This result stresses the fact that the π-calculus is a good compromise between expressiveness and simplicity. As a further contribution, we also prove that the polyadic π-calculus is strictly more expressive than the monadic one. A posteriori, this fact justifies the use of type-systems [19, 27, 24] to obtain a fragment of the former calculus that can be reasonably translated in the latter one.

This paper is organised as follows. In Section 2, we present a family of eight π-based asynchronous calculi arising from the combination of the three features studied. In Section 3, we present the criteria an encoding should satisfy to be a reasonable means for language comparison; there, we also sum-up the results of the paper, that are proved in Sections 4 and 5. We start with the encodability results and then we present the impossibility results, that are the main contribution of our work. Finally, in Section 6, we conclude the paper by also touching upon related work.

2 A Family of π-Based Calculi

As we said in the Introduction, we shall assess the expressiveness of the communication primitives studied by putting them in a common framework, inspired by the asynchronous π-calculus. We assume two disjoint and countable sets: *names*, $\mathcal{N}$, ranged over by $a, b, x, y, n, m, \cdots$, and *process variables*, $\mathcal{X}$, ranged over by $X, Y, \cdots$.

Notationally, when a name is used as a channel, we shall prefer letters $a, b, c, \cdots$; when a name is used as an input variable, we shall prefer letters $x, y, z, \cdots$; to denote a generic name, we shall use letters $n, m, \cdots$. The (parametric) syntax of our calculi is

$$\begin{array}{rcl} P, Q, R & ::= & \mathbf{0} \;\big|\; OUT \;\big|\; IN.P \;\big|\; (\nu n)P \;\big|\; P|Q \\ & \big| & \textbf{if } n = m \textbf{ then } P \textbf{ else } Q \;\big|\; \textbf{rec}\, X.P \;\big|\; X \end{array}$$

The different calculi will be obtained by plugging into this basic syntax a proper definition for input (IN) and output (OUT) actions. As usual, $\mathbf{0}$ and $P|Q$ denote the terminated process and the parallel composition of two processes, while $(\nu n)P$ restricts to P the visibility of n; finally, **if** $n = m$ **then** P **else** Q, **rec** $X.P$ and X are the standard constructs for conditional evolution, process definition and process invocation.

In this paper, we study the possible combinations of three features for asynchronous communications: *arity* (monadic vs. polyadic data), *communication medium* (channels vs. shared dataspaces) and *pattern-matching*. As a result, we have a family of eight calculi, denoted as $\Pi_{a,m,p}$, whose generic element is denoted as $\pi_{\beta_1\beta_2\beta_3}$, where $\beta_i \in \{0, 1\}$. Intuitively, $\beta_1 = 1$ iff we have polyadic data, $\beta_2 = 1$ iff we have channel-based communications and $\beta_3 = 1$ iff we have pattern-matching. Thus, the full syntax of every calculus is obtained from the following productions:

$$\begin{array}{llll}
\pi_{000}: & P, Q, R ::= \ldots & IN ::= (x) & OUT ::= \langle b \rangle \\
\pi_{001}: & P, Q, R ::= \ldots & IN ::= (T) & OUT ::= \langle b \rangle \\
\pi_{010}: & P, Q, R ::= \ldots & IN ::= a(x) & OUT ::= \overline{a}\langle b \rangle \\
\pi_{011}: & P, Q, R ::= \ldots & IN ::= a(T) & OUT ::= \overline{a}\langle b \rangle \\
\pi_{100}: & P, Q, R ::= \ldots & IN ::= (\widetilde{x}) & OUT ::= \langle \widetilde{b} \rangle \\
\pi_{101}: & P, Q, R ::= \ldots & IN ::= (\widetilde{T}) & OUT ::= \langle \widetilde{b} \rangle \\
\pi_{110}: & P, Q, R ::= \ldots & IN ::= a(\widetilde{x}) & OUT ::= \overline{a}\langle \widetilde{b} \rangle \\
\pi_{111}: & P, Q, R ::= \ldots & IN ::= a(\widetilde{T}) & OUT ::= \overline{a}\langle \widetilde{b} \rangle
\end{array}$$

where

$$T \;::=\; x \;\big|\; \ulcorner n \urcorner \qquad\qquad (\textit{Template})$$

and $\widetilde{\cdot}$ denotes a (possibly empty) sequence of elements of kind $\cdot$ (whenever useful, we shall write a tuple $\widetilde{\cdot}$ as the sequence of its elements, separated by a comma). Template fields of kind x are called *formal* and can be replaced by every name upon withdrawal of a datum; template fields of kind $\ulcorner n \urcorner$ are called *actual* and impose that the datum withdrawn contains exactly name n.

$\Pi_{a,m,p}$ can be easily ordered by language containment; in particular, $\pi_{\beta_1\beta_2\beta_3}$ can be seen as a sub-language of $\pi_{\beta'_1\beta'_2\beta'_3}$ if and only if, for every $i \in \{1, 2, 3\}$, it holds that $\beta_i \leq \beta'_i$. As an extremal example, consider π_{000} and π_{111}: monadic data are a particular case of polyadic data (all of length one); a shared dataspace can be modelled by letting all communications happen on the same global channel, say `ether`; finally, absence of pattern-matching can be obtained by only considering templates without actual fields.

Notice that π_{010} and π_{110} are very similar to the (monadic/polyadic) asynchronous π-calculus [17, 4]; π_{101} relies on the communication paradigm adopted in LINDA [15]; π_{000} and π_{100} rely on the communication paradigm adopted in the (monadic/polyadic) Ambient Calculus [8]; π_{001} and π_{011} rely on the communication paradigm adopted in LCKLAIM and CKLAIM [11], respectively; finally, π_{111} relies on the communication paradigm adopted, e.g., in μKLAIM [11] or in semantic-π [9].

As usual, $a(\cdots, x, \cdots).P$ and $(\nu x)P$ bind x in P, while $\mathbf{rec}\, X.P$ binds X in P. The corresponding notions of free and bound names of a process, $\mathrm{FN}(P)$ and $\mathrm{BN}(P)$, and of alpha-conversion, $=_\alpha$, are assumed. We let $\mathrm{N}(P)$ denote $\mathrm{FN}(P) \cup \mathrm{BN}(P)$.

Operational semantics. The operational semantics of the calculi is given by means of a *labelled transition system* (LTS) describing the actions a process can perform to evolve. Judgements take the form $P \xrightarrow{\alpha} P'$, meaning that P can become P' upon execution of α. *Labels* take the form

$$\alpha \ ::= \ \tau \ \big| \ a?\widetilde{b} \ \big| \ (\nu\widetilde{c})a!\widetilde{b} \ \big| \ ?\widetilde{b} \ \big| \ (\nu\widetilde{c})!\widetilde{b}$$

Traditionally, τ denotes an internal computation; $a?\widetilde{b}$ and $(\nu\widetilde{c})a!\widetilde{b}$ denote the reception/sending of a sequence of names $\widetilde{b}$ along channel a; when channels are not present (namely, in π_{-0-}), $?\widetilde{b}$ and $(\nu\widetilde{c})!\widetilde{b}$ denote the withdrawal/emission of $\widetilde{b}$ from/in the shared dataspace. In $(\nu\widetilde{c})a!\widetilde{b}$ and $(\nu\widetilde{c})!\widetilde{b}$, some of the sent names, viz. $\widetilde{c}$ ($\subseteq \widetilde{b}$), are restricted. Notationally, $(\nu\widetilde{c})\text{-}!\widetilde{b}$ stands for either $(\nu\widetilde{c})a!\widetilde{b}$ or $(\nu\widetilde{c})!\widetilde{b}$. As usual, $\mathrm{BN}((\nu\widetilde{c})\text{-}!\widetilde{b}) \triangleq \widetilde{c}$; $\mathrm{FN}(\alpha)$ and $\mathrm{N}(\alpha)$ are defined accordingly.

The LTS provides some rules shared by all the calculi; the different semantics are obtained from the axioms for input/output actions. The common rules, reported below, are an easy adaptation of an early-style LTS for the π-calculus; thus, we do not comment them and refer the interested reader to [23].

$$\frac{P \xrightarrow{?\widetilde{b}} P' \qquad Q \xrightarrow{!\widetilde{b}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad\qquad \frac{P \xrightarrow{a?\widetilde{b}} P' \qquad Q \xrightarrow{a!\widetilde{b}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\frac{P \xrightarrow{\alpha} P' \qquad n \notin \mathrm{N}(\alpha)}{(\nu n)P \xrightarrow{\alpha} (\nu n)P'} \qquad\qquad \frac{P \xrightarrow{(\nu\widetilde{c})\text{-}!\widetilde{b}} P' \qquad n \in \mathrm{FN}(\widetilde{b}) \setminus \{\text{-}, \widetilde{c}\}}{(\nu n)P \xrightarrow{(\nu n, \widetilde{c})\text{-}!\widetilde{b}} P'}$$

$$\frac{P \xrightarrow{\alpha} P' \qquad \mathrm{BN}(\alpha) \cap \mathrm{FN}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad\qquad \frac{P \equiv P_1 \xrightarrow{\alpha} P_2 \equiv P'}{P \xrightarrow{\alpha} P'}$$

The structural equivalence, $\equiv$, rearranges a process to let it evolve according to the rules of the LTS. Its defining axioms are the standard π-calculus' ones [23]:

$$P \mid \mathbf{0} \equiv P \qquad\qquad P \mid Q \equiv Q \mid P \qquad\qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$\mathbf{if}\ n = n\ \mathbf{then}\ P\ \mathbf{else}\ Q \equiv P \qquad\qquad \mathbf{if}\ n = m\ \mathbf{then}\ P\ \mathbf{else}\ Q \equiv Q \ \text{ if } n \neq m$$

$$P \equiv P' \text{ if } P =_\alpha P' \qquad (\nu n)\mathbf{0} \equiv \mathbf{0} \qquad (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$$

$$P \,|\, (\nu n)Q \equiv (\nu n)(P \,|\, Q) \text{ if } n \notin \mathrm{FN}(P) \qquad \mathbf{rec}\, X.P \equiv P\{^{\mathbf{rec}\, X.P}/_X\}$$

To define the semantics for the basic actions of the various calculi, we must specify when a template matches a datum. Intuitively, this happens whenever both have the same length and corresponding fields match (i.e., $\ulcorner n \urcorner$ matches n and x matches every name). This can be formalised via a partial function, called *pattern-matching* and written MATCH, that also returns a substitution σ; the latter will be applied to the process that performed the input to replace template formal fields with the corresponding names of the datum retrieved. These intuitions are formalised by the following rules:

$$\mathrm{MATCH}(\,;\,) = \epsilon \qquad \mathrm{MATCH}(\ulcorner n \urcorner; n) = \epsilon \qquad \mathrm{MATCH}(x; n) = \{^n/_x\}$$

$$\frac{\mathrm{MATCH}(T; b) = \sigma_1 \qquad \mathrm{MATCH}(\widetilde{T}; \widetilde{b}) = \sigma_2}{\mathrm{MATCH}(T, \widetilde{T}\,;\, b, \widetilde{b}) = \sigma_1 \circ \sigma_2}$$

where 'ϵ' denotes the empty substitution and '$\circ$' denotes substitution composition. Now, the operational rules for output/input actions in calculi $\pi_{\text{-}0\text{-}}$ are

$$\langle \widetilde{b} \rangle \xrightarrow{!\widetilde{b}} \mathbf{0} \qquad (\widetilde{T}).P \xrightarrow{?\widetilde{b}} P\sigma \text{ if } \mathrm{MATCH}(\widetilde{T}\,;\,\widetilde{b}) = \sigma$$

and, similarly, the rules for calculi $\pi_{\text{-}1\text{-}}$ are

$$\overline{a}\langle \widetilde{b} \rangle \xrightarrow{a!\widetilde{b}} \mathbf{0} \qquad a(\widetilde{T}).P \xrightarrow{a?\widetilde{b}} P\sigma \text{ if } \mathrm{MATCH}(\widetilde{T}\,;\,\widetilde{b}) = \sigma$$

Notation. A substitution σ is a finite partial mapping of names for names; $P\sigma$ denotes the (capture avoiding) application of σ to P. As usual, we let $\Rightarrow$ stand for $\xrightarrow{\tau}{}^{*}$ (i.e., the reflexive and transitive closure of $\xrightarrow{\tau}$) and $\xRightarrow{\alpha}$ stand for $\Rightarrow\xrightarrow{\alpha}\Rightarrow$. We shall write $P \xrightarrow{\alpha}$ to mean that there exists a process P' such that $P \xrightarrow{\alpha} P'$; a similar notation is adopted for $P \Rightarrow$ and $P \xRightarrow{\alpha}$. Moreover, we let ϕ range over visible actions (i.e. labels different from τ) and ρ to range over (possibly empty) sequences of visible actions. Formally, $\rho ::= \varepsilon \mid \phi \cdot \rho$, where '$\varepsilon$' denotes the empty sequence of actions and '$\cdot$' represents concatenation; then, $N \xRightarrow{\varepsilon}$ is defined as $N \Rightarrow$ and $N \xRightarrow{\phi\cdot\rho}$ is defined as $N \xRightarrow{\phi}\xRightarrow{\rho}$.

3 Quality of an Encoding and Overview of Our Results

We now study the relative expressive power of the calculi in $\Pi_{a,m,p}$ by trying to encode one in another. Formally, an *encoding* $[\![\cdot]\!]$ is a function mapping terms of the source language into terms of the target language. As pointed out elsewhere [26, 10, 22], the relative expressive power of our calculi can be established by defining some criteria to evaluate the quality of the encodings or to prove impossibility results.

The main requirement, that we call *faithfulness*, is that the encoding must not change the semantics of a source term, i.e. it must preserve the observable behaviour of the term

without introducing new behaviours. As very clearly discussed in [21], there are several ways to formalise this idea; we shall define it in the simplest possible way, by means of *barbs* and *divergence*.

Definition 1 (Barbs and Divergence). *P offers a* barb*, written $P \Downarrow$, iff $P \xRightarrow{(\nu\widetilde{c})\,\overline{\ }!\widetilde{b}}$.* *$P$* diverges*, written $P \Uparrow$, iff $P \xrightarrow{\tau}{}^{\omega}$.*

The idea is to identify a basic observable behaviour (or *barb*) for the languages considered and require that the encoding preserves and reflects it (i.e., the encoding should maintain all the original barbs without introducing new ones). In the setting of an asynchronous language [2, 6], a barb is the possibility of emitting some datum.[1] Since barb preservation and reflection alone are too weak, it is also required that the computations of a process correspond to the computations of its encoding, and vice versa; this property is usually known as *operational correspondence*. Barb preservation and operational correspondence together yield *(weak) barbed bisimulation* [20, 2] that, however, is insensitive to divergence (i.e., it can equate a term with an infinite computation and a term with only finite computations). In our setting, it is clearly undesirable to have an encoding that turns a terminating term into a divergent one, since this would change the behaviour of the source term. So, we need a further requirement stating that also divergence must be preserved and reflected by the encoding.

Finally, a good encoding cannot depend on the particular names involved in the source process, since we are dealing with a family of name-passing calculi; we call this property *name invariance*. Furthermore, the encoding should not decrease the degree of parallelism in favour of centralised entities that control the behaviour of the encoded term: if we can find some process behaviour that cannot be implemented in the target language with the same degree of distribution as in the source one, then surely the former language will be "weaker" than the latter one. We express this last property as *homomorphism w.r.t.* '$|$'.

To sum up, we consider an encoding as a *'reasonable'* means to compare the expressive power of two languages if it enjoys all the properties discussed so far.

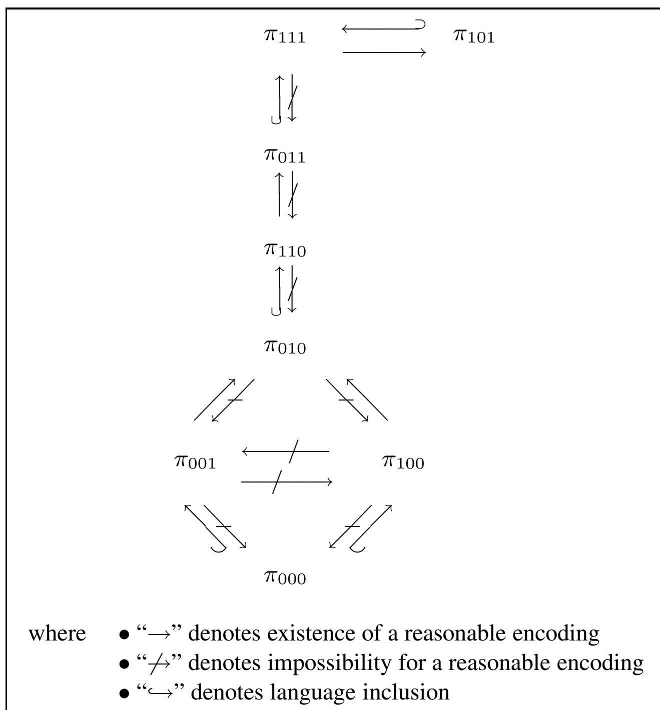
Definition 2 (Reasonable Encoding). *An encoding $[\![\cdot]\!]$ is* reasonable *if it enjoys the following properties:*

1. (homomorphism w.r.t. '$|$'): $[\![P_1|P_2]\!] \triangleq [\![P_1]\!] \mid [\![P_2]\!]$
2. (name invariance): $[\![P\sigma]\!] \triangleq [\![P]\!]\sigma$*, for every permutation of names* σ
3. (faithfulness): $P \Downarrow$ *iff* $[\![P]\!] \Downarrow$*;* $P \Uparrow$ *iff* $[\![P]\!] \Uparrow$
4. (operational correspondence):
   (a) if $P \xrightarrow{\tau} P'$ *then* $[\![P]\!] \xRightarrow{\tau} [\![P']\!]$
   (b) if $[\![P]\!] \xrightarrow{\tau} Q$ *then there exists a* P' *such that* $P \Rightarrow P'$ *and* $Q \Rightarrow [\![P']\!]$

The results of our paper are summarised in Table 1. It is worth noting that all the languages are Turing complete: it is easy to show that π_{000} can encode (by introducing

[1] We choose here a very weak form of barbs. This fact strengthens our impossibility results; on the other hand, our possibility results are not undermined by this choice, since they would also enjoy properties expressed in terms of more significant barbs, such as those in [2, 6].

Table 1. Overview of the Results



divergence[2]) π_{001} that, in turn, is Turing complete (actually, the fragment without template formal fields suffices, see [6]). Moreover, notice that Definition 2(4).b is a weak form of correspondence; this makes our impossibility results stronger. However, for the encodability results, a better definition should keep into account every possible computation $[\![P]\!] \Rightarrow Q$. With this stronger property, the encodings provided in Sections 4.2 and 4.4 would not enjoy operational correspondence; we leave for future work the task of establishing whether encodings of π_{001} and π_{110} in π_{010} and π_{011} satisfying this stronger property exist or not.

4 Encodability Results

We start with the positive results, i.e. the "$\rightarrow$" arrows of Table 1. In all the cases, we shall describe only the translation of the input and output actions; the remaining operators will be translated homomorphically (this trivially satisfies Definition 2(1)). Moreover, in what follows we are going to prove only that the encodings do not introduce divergence; preservation of divergence is a trivial consequence of Definition 2(4).a; Definition 2(2) and barb preservation/reflection will hold by construction of the encodings; Definition 2(4) can be routinely proved.

[2] To study Turing completeness, the fact that the encoding introduces divergence is irrelevant.

4.1 An Encoding of π_{111} in π_{101}

The only feature of π_{111} not present in π_{101} is the possibility of specifying the name of a channel where the exchange happens. However, thanks to pattern-matching, this feature can be very easily encoded in π_{101}: it suffices to impose that the first name of every datum represents the name of the channel where the interaction is scheduled and that every template starts with the corresponding actual field. This discipline is rendered by the following encoding:

$$[\![\, \overline{a}\langle \widetilde{b}\rangle \,]\!] \triangleq \langle a, \widetilde{b}\rangle$$

$$[\![\, a(\widetilde{T}).P \,]\!] \triangleq (\ulcorner a \urcorner, \widetilde{T}).[\![\, P \,]\!]$$

Proposition 1. *The encoding* $[\![\,\cdot\,]\!] : \pi_{111} \longrightarrow \pi_{101}$ *is reasonable.*

Proof. Definition 2(2) holds by construction. Definition 2(4).b can be proved as a stronger claim: if $[\![\, P \,]\!] \xrightarrow{\tau} Q$, then $Q \equiv [\![\, P' \,]\!]$ and $P \xrightarrow{\tau} P'$ (this result, like Definition 2(4).a, is proved by an easy induction over the shortest inference for $\xrightarrow{\tau}$). Definition 2(3) holds easily; just notice that the stronger formulation of operational correspondence mentioned above implies that the encoding cannot introduce divergence. □

4.2 An Encoding of π_{001} in π_{010}

We now have to translate the monadic pattern-matching of π_{001} into the channel-based exchanges of π_{010}. This would have been an easy task, if only actual fields occurred in templates: indeed, $\langle b\rangle$ would have been translated in $\overline{b}\langle b\rangle$ and, correspondingly, $(\ulcorner b \urcorner).P$ would have been translated in $b(y).[\![\, P \,]\!]$, for y fresh. This encoding, however, does not work well when trying to translate $(x).P$.

Thus, $\langle b\rangle$ in π_{001} should correspond to two outputs in π_{010}: one over b, to mimic name matching as described above, and one over a fresh and reserved channel ether, to enable inputs with formal fields. Symmetrically, an input action is translated in two successive inputs: the first one from ether and the second one from the received value, if we are translating an input with a formal field, and vice versa, otherwise. For example, $\langle b\rangle \mid \langle c\rangle \mid (\ulcorner c \urcorner).P$ is translated to $\overline{b}\langle b\rangle \mid \overline{\mathtt{ether}}\langle b\rangle \mid \overline{c}\langle c\rangle \mid \overline{\mathtt{ether}}\langle c\rangle \mid c(y).\mathtt{ether}(z).[\![\, P \,]\!]$. We believe that this encoding is reasonable, but proving that it does not introduce divergence is hard because of the possible interferences between parallel components (e.g., the above example could evolve in $\overline{b}\langle b\rangle \mid \overline{\mathtt{ether}}\langle c\rangle \mid [\![\, P \,]\!]$, by performing a communication between $c(y)$ and $\overline{c}\langle c\rangle$ and between $\mathtt{ether}(z)$ and $\overline{\mathtt{ether}}\langle b\rangle$).

The problem is that the two outputs in the translation of $\langle b\rangle$ are totally unrelated. This can be fixed by associating every output with a restricted name and by using such a name to reduce the effects of interferences. Formally,

$$[\![\, \langle b\rangle \,]\!] \triangleq (\nu n)(\overline{\mathtt{ether}}\langle n\rangle \mid \overline{b}\langle n\rangle \mid \overline{n}\langle b\rangle)$$

$$[\![\, (x).P \,]\!] \triangleq \mathtt{ether}(y).y(x).x(z).\mathbf{if}\ y = z\ \mathbf{then}\ [\![\, P \,]\!]$$

$$[\![\, (\ulcorner b \urcorner).P \,]\!] \triangleq b(y).y(y').\mathtt{ether}(z).\mathbf{if}\ y = z\ \mathbf{then}\ [\![\, P \,]\!]$$

for n, y, y' and z fresh names. Clearly, this solution does not rule out interferences; it simply blocks interfering processes. This suffices to make the proof of reasonableness easier; to this aim, the key result is the following Lemma.

Lemma 1. *Let κ be the number of top-level outputs in P. If $[\![P]\!] \xrightarrow{\tau}{}^{3\kappa+1} Q$, then there exists a P' such that $P \xrightarrow{\tau} P'$ and $[\![P]\!] \xrightarrow{\tau}{}^{3} [\![P']\!] \Rightarrow Q$.*

Proposition 2. *The encoding $[\![\cdot]\!] : \pi_{001} \longrightarrow \pi_{010}$ is reasonable.*

Proof. We only prove that $[\![\cdot]\!]$ does not introduce divergence; the other requirements are simple. Assume that $[\![P]\!] \Uparrow$, i.e. there exists an infinite computation $[\![P]\!] \triangleq Q_0 \xrightarrow{\tau} Q_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} Q_{3\kappa+1} \xrightarrow{\tau} \cdots$. By Lemma 1, there exists a P' such that $[\![P]\!] \xrightarrow{\tau}{}^{3} [\![P']\!] \Rightarrow Q_{3\kappa+1}$ and $P \xrightarrow{\tau} P'$. But $[\![P']\!]$ is still divergent; indeed, $[\![P']\!] \Rightarrow Q_{3\kappa+1} \xrightarrow{\tau} \cdots$. By iterating this reasoning, we can build up a divergent computation for P, i.e. $P \xrightarrow{\tau} P' \xrightarrow{\tau} \cdots$; hence, $[\![\cdot]\!]$ does not introduce divergence. □

4.3 An Encoding of π_{100} in π_{010}

The only feature of π_{100} is that it can check the arity of a datum before retrieving it (see the definition of function MATCH). This, however, can be mimicked by the channel-based communication of π_{010}. Indeed, we assume a (reserved) channel for every possible arity: a datum of arity k will be represented as an output over channel $\mathtt{k}$; an input of arity k will be represented as an input from $\mathtt{k}$; a communication over $\mathtt{k}$ in π_{010} can happen if and only if pattern-matching succeeds in π_{100}; finally, the exchanged datum is a restricted name that will be used in the actual data exchange.

The encoding assumes that $\mathtt{0}, \mathtt{1}, \ldots, \mathtt{k}, \ldots$ are fresh and reserved names; then

$$[\![\langle b_1, \cdots, b_k \rangle]\!] \triangleq (\nu n)(\overline{\mathtt{k}}\langle n \rangle \mid n(y).(\overline{y}\langle b_1 \rangle \mid n(y).(\overline{y}\langle b_2 \rangle \mid n(y).(\cdots n(y).\overline{y}\langle b_k \rangle) \cdots)))$$

$$[\![(x_1, \cdots, x_k).P]\!] \triangleq \mathtt{k}(x).(\nu m)(\overline{x}\langle m \rangle \mid m(x_1).(\overline{x}\langle m \rangle \mid m(x_2).(\cdots (\overline{x}\langle m \rangle \mid m(x_k).[\![P]\!]) \cdots)))$$

for n, m, x and y fresh names. The datum emitted over $\mathtt{k}$ (viz. n) is used as a "synchroniser", to keep the order of the transmitted data and force the right name-to-variable association. The actual exchange takes place over a restricted channel created by the receiver (viz. m) and transmitted along n as an ack to the sender.

Like before, reasonableness of this encoding can be proved by using the following Lemma. Moreover, notice that for this encoding the stronger version of Definition 2(4).b holds: if $[\![P]\!] \Rightarrow Q$, then there is a P' such that $P \Rightarrow P'$ and $Q \Rightarrow [\![P']\!]$.

Lemma 2. *Let κ and λ be the number and the maximum arity of top-level outputs in P, respectively. If $[\![P]\!] \xrightarrow{\tau}{}^{\kappa(2\lambda+1)} Q$, then there exists a P' such that $P \xrightarrow{\tau} P'$ and $[\![P]\!] \xrightarrow{\tau}{}^{2h+1} [\![P']\!] \Rightarrow Q$, for $0 \leq h \leq \lambda$.*

Proposition 3. *The encoding $[\![\cdot]\!] : \pi_{100} \longrightarrow \pi_{010}$ is reasonable.*

4.4 An Encoding of π_{110} in π_{011}

In π_{110}, a communication succeeds if (and only if) a datum of a proper length is present over the channel specified by the inputting process. So, two kinds of information are atomically verified: the length of the message and the channel where it should be transmitted. This can be mimicked in π_{011} by having one fresh and reserved name for every length (say, $\mathtt{0}, \mathtt{1}, \ldots, \mathtt{k}, \ldots$); a k-ary input from a is then translated into a process starting with $a(\ulcorner \mathtt{k} \urcorner)$ and, correspondingly, a k-ary output on a is translated into a process offering $\mathtt{k}$ at a. Once this communication took place, we are sure that a k-ary datum is available on a; we then proceed similarly to Section 4.3 for the actual data exchange: a new channel n is made available on a, to maintain the order of messages, while a new channel m is sent back on n to transmit the datum name by name.

However, we need to enhance the encoding of Section 4.3 to avoid interferences due to the fact that the existence of a k-ary output and the acquisition of the new name for the actual exchange are not atomic here. Indeed, the translation of $\overline{a}\langle b_1, b_2 \rangle \mid \overline{a}\langle c_1, c_2, c_3 \rangle \mid a(x_1, x_2).P \mid a(x_1, x_2, x_3).Q$ can originate interferences that can lead to divergence. Thus, like in Section 4.2, we shall verify at the end of the data exchange the consistency of the exchange, i.e. that a k-ary data has really been retrieved. To this aim, let $\mathtt{end}$ be another fresh and reserved name; then

$$
\begin{aligned}
[\![\, \overline{a}\langle b_1, \cdots, b_k \rangle \,]\!] &\triangleq \overline{a}\langle \mathtt{k} \rangle \mid (\nu n)(\overline{a}\langle n \rangle \mid n(y).(\overline{y}\langle b_1 \rangle \mid \cdots\ n(y).(\overline{y}\langle b_k \rangle \mid \\
&\qquad\qquad n(y).\overline{y}\langle \mathtt{end} \rangle) \cdots)) \\
[\![\, a(x_1, \cdots, x_k).P \,]\!] &\triangleq a(\ulcorner \mathtt{k} \urcorner).a(x).(\nu m)(\overline{x}\langle m \rangle \mid m(x_1).(\cdots \mid (\overline{x}\langle m \rangle \mid m(x_k). \\
&\qquad\qquad (\overline{x}\langle m \rangle \mid m(\ulcorner \mathtt{end} \urcorner).[\![\, P \,]\!]) \cdots)))
\end{aligned}
$$

for n, m, x and y fresh names. Reasonableness of this encoding can be proved like in Proposition 2, as a consequence of the following Lemma.

Lemma 3. *Let κ and λ be the number and the maximum arity of top-level outputs in P, respectively. If $[\![\, P \,]\!] \xrightarrow{\tau}{}^{\kappa(2\lambda+3)+1} Q$, then there exists a P' such that $P \xrightarrow{\tau} P'$ and $[\![\, P \,]\!] \xrightarrow{\tau}{}^{2h+4} [\![\, P' \,]\!] \Rightarrow Q$, for $0 \leq h \leq \lambda$.*

Proposition 4. *The encoding $[\![\cdot]\!] : \pi_{110} \longrightarrow \pi_{011}$ is reasonable.*

5 Impossibility Results

We now consider the impossibility results, i.e. the "$\not\to$" arrows of Table 1, that are the main technical contribution of this paper. They are all proved by contradiction: we assume that a reasonable encoding exists and show that it introduces divergence. Often, the contradiction is obtained by exhibiting a process that cannot reduce but whose encoding reduces. This fact, together with operational correspondence, implies that the encoding introduces divergence, as stated by the following simple result.

Proposition 5. *Let $[\![\cdot]\!]$ be an operationally corresponding encoding. If there exists a process P such that $P \stackrel{\tau}{\not\longrightarrow}$ but $[\![\, P \,]\!] \xrightarrow{\tau}$, then $[\![\cdot]\!]$ introduces divergence.*

Proof. The fact that $\llbracket P \rrbracket \xrightarrow{\tau} Q$ implies, by operational correspondence, that $P \Rightarrow P'$, for some P' such that $Q \Rightarrow \llbracket P' \rrbracket$. But the only P' such that $P \Rightarrow P'$ is P itself; thus, $\llbracket P \rrbracket \xrightarrow{\tau}{}^{+} \llbracket P \rrbracket$, i.e. $\llbracket P \rrbracket$ diverges. □

Theorem 1. *There exists no reasonable encoding of* π_{011} *in* π_{110}.

Proof. Assume that $\llbracket \cdot \rrbracket$ is reasonable and consider the process $a(\ulcorner b \urcorner) \mid \overline{a}\langle b \rangle$, for $a \neq b$, that evolves in $\mathbf{0}$. By operational correspondence, $\llbracket a(\ulcorner b \urcorner) \mid \overline{a}\langle b \rangle \rrbracket \Rightarrow \llbracket \mathbf{0} \rrbracket$; moreover, by faithfulness, $\llbracket a(\ulcorner b \urcorner) \rrbracket \not\Downarrow$, $\llbracket \overline{a}\langle b \rangle \rrbracket \Downarrow$ and $\llbracket \mathbf{0} \rrbracket \not\Downarrow$. Thus, the barb of $\llbracket \overline{a}\langle b \rangle \rrbracket$ must be consumed in the computation leading $\llbracket a(\ulcorner b \urcorner) \mid \overline{a}\langle b \rangle \rrbracket$ to $\llbracket \mathbf{0} \rrbracket$.

Now, notice that $\llbracket \overline{a}\langle b \rangle \rrbracket$ cannot perform a τ-step otherwise, by Proposition 5, $\llbracket \cdot \rrbracket$ would introduce divergence. This fact, together with $\llbracket a(\ulcorner b \urcorner) \mid \overline{a}\langle b \rangle \rrbracket \triangleq \llbracket a(\ulcorner b \urcorner) \rrbracket \mid \llbracket \overline{a}\langle b \rangle \rrbracket$, implies that $\llbracket a(\ulcorner b \urcorner) \rrbracket$ consumed the barb offered by $\llbracket \overline{a}\langle b \rangle \rrbracket$. Thus, it must be that $\llbracket a(\ulcorner b \urcorner) \rrbracket \xrightarrow{n?\widetilde{c}}$, for some n and $\widetilde{c}$ such that $\llbracket \overline{a}\langle b \rangle \rrbracket \xrightarrow{(\nu\widetilde{c'})n!\widetilde{c}}$; then $\llbracket a(\ulcorner b \urcorner) \rrbracket \xrightarrow{n?\widetilde{d}}$, for every $\widetilde{d}$ of the same arity as $\widetilde{c}$, i.e. $|\widetilde{d}| = |\widetilde{c}|$.

If $n \neq b$, then pick up $e \notin \{a, b, n\}$ and the permutation of names swapping b and e; by name invariance, it holds that $\llbracket \overline{a}\langle e \rangle \rrbracket \xrightarrow{(\nu\widetilde{f'})n!\widetilde{f}}$, where $\widetilde{f}$ and $\widetilde{f'}$ are the renaming of $\widetilde{c}$ and $\widetilde{c'}$. In particular, $|\widetilde{c}| = |\widetilde{f}|$. Then, $\llbracket a(\ulcorner b \urcorner) \mid \overline{a}\langle e \rangle \rrbracket \xrightarrow{\tau}$, while $a(\ulcorner b \urcorner) \mid \overline{a}\langle e \rangle \not\xrightarrow{\tau}$. By Proposition 5, $\llbracket \cdot \rrbracket$ is not reasonable, as it introduces divergence; contradiction.

If $n = b$, then pick up $e \notin \{a, b\}$, the permutation of names swapping a and e, and work like before, with process $\llbracket a(\ulcorner b \urcorner) \mid \overline{e}\langle b \rangle \rrbracket$. □

Corollary 1. *There exists no reasonable encoding of* π_{001} *in* π_{100}.

Proof. Trivial consequence of Theorem 1, since π_{001} and π_{100} can be seen as the sub-calculi of π_{011} and π_{110} where all the communications happen on the same (unique and global) channel. □

Theorem 2. *There exists no reasonable encoding of* π_{010} *in* π_{100}.

Proof. The proof is similar to that of Theorem 1. We start with process $\overline{a}\langle b \rangle \mid a(x)$, for $a \neq b$; it holds that $\llbracket \overline{a}\langle b \rangle \rrbracket \xrightarrow{(\nu\widetilde{c'})!\widetilde{c}}$ and $\llbracket a(x) \rrbracket \xrightarrow{?\widetilde{c}}$, for some $\widetilde{c}$. By name invariance, $\llbracket \overline{b}\langle a \rangle \rrbracket \xrightarrow{(\nu\widetilde{d'})!\widetilde{d}}$, where $\widetilde{d}$ and $\widetilde{d'}$ are obtained by swapping a and b in $\widetilde{c}$ and $\widetilde{c'}$; thus, $|\widetilde{d}| = |\widetilde{c}|$. Now, trivially, $\llbracket a(x) \rrbracket \xrightarrow{?\widetilde{d}}$ and $\llbracket \overline{b}\langle a \rangle \mid a(x) \rrbracket \xrightarrow{\tau}$, while $\overline{b}\langle a \rangle \mid a(x) \not\xrightarrow{\tau}$. □

Theorem 3. *There exists no reasonable encoding of* π_{110} *in* π_{010}.

Proof. Similarly to the proof of Theorem 1, consider the process $a(x, y) \mid \overline{a}\langle b, c \rangle$; again, $\llbracket \overline{a}\langle b, c \rangle \rrbracket \xrightarrow{(\nu\widetilde{d'})n!d}$ and $\llbracket a(x, y) \rrbracket \xrightarrow{n?d}$, for some d and $\widetilde{d'}$. If $n \neq a$, choose $e \neq a$; by name invariance, $\llbracket \overline{e}\langle b, c \rangle \rrbracket \xrightarrow{(\nu\widetilde{f'})n!f}$ and $\llbracket a(x, y) \mid \overline{e}\langle b, c \rangle \rrbracket \xrightarrow{\tau}$, while $a(x, y) \mid \overline{e}\langle b, c \rangle \not\xrightarrow{\tau}$. If $n = a$, consider $a(x, y, z) \mid \overline{a}\langle b, c, c \rangle$; like before, $\llbracket \overline{a}\langle b, c, c \rangle \rrbracket \xrightarrow{(\nu\widetilde{e'})m!e}$ and $\llbracket a(x, y, z) \rrbracket \xrightarrow{m?e}$. Then, if $m = a$, we have that $\llbracket a(x, y) \mid \overline{a}\langle b, c, c \rangle \rrbracket \xrightarrow{\tau}$; otherwise, choose $f \neq a$ and conclude that $\llbracket a(x, y, z) \mid \overline{f}\langle b, c, c \rangle \rrbracket \xrightarrow{\tau}$. □

Corollary 2. *There exist no reasonable encodings of* π_{001} *and* π_{100} *in* π_{000}.

Proof. Easily derivable from Corollary 1 and Theorem 3, respectively. □

Theorem 4. *There exists no reasonable encoding of* π_{111} *in* π_{011}.

Proof. Consider the process $\overline{a}\langle b,c\rangle \mid a(\ulcorner b\urcorner,\ulcorner c\urcorner)$, for a, b and c pairwise distinct. Like in Theorem 1, we have that $[\![\, a(\ulcorner b\urcorner,\ulcorner c\urcorner) \,]\!] \xrightarrow{n?m}$ and $[\![\, \overline{a}\langle b,c\rangle \,]\!] \xrightarrow{(\nu\widetilde{m})n!m}$. If the input of $[\![\, a(\ulcorner b\urcorner,\ulcorner c\urcorner) \,]\!]$ has been generated by relying on a template formal field, then $[\![\, a(\ulcorner b\urcorner,\ulcorner c\urcorner) \,]\!] \xrightarrow{n?l}$, for every l; by Proposition 5, this would suffice to build up a divergent computation for $[\![\, a(\ulcorner b\urcorner,\ulcorner c\urcorner) \mid \overline{a}\langle b,d\rangle \,]\!]$, for every $d \neq c$. Otherwise, the input of $[\![\, a(\ulcorner b\urcorner,\ulcorner c\urcorner) \,]\!]$ relies on an actual field; we then consider the following possibilities for n and m:

1. $c \notin \{n,m\}$: let $d \neq c$ and consider the permutation that swaps c and d; then, $[\![\, \overline{a}\langle b,d\rangle \,]\!] \xrightarrow{(\nu\widetilde{m})n!m}$ and $[\![\, \overline{a}\langle b,d\rangle \mid a(\ulcorner b\urcorner,\ulcorner c\urcorner) \,]\!] \xrightarrow{\tau}$.
2. $n = c$, $m \neq b$: let $d \neq b$ and consider the permutation that swaps b and d; like before, $[\![\, \overline{a}\langle d,c\rangle \mid a(\ulcorner b\urcorner,\ulcorner c\urcorner) \,]\!] \xrightarrow{\tau}$.
3. $n = c$, $m = b$: let $d \neq a$ and consider the permutation that swaps a and d; then, $[\![\, \overline{d}\langle b,c\rangle \mid a(\ulcorner b\urcorner,\ulcorner c\urcorner) \,]\!] \xrightarrow{\tau}$.
4. $m = c, n \neq b$: like case 2.
5. $m = c, n = b$: like case 3. □

Theorem 5. *There exists no reasonable encoding of* π_{010} *in* π_{001}.

Proof. By contradiction, assume that there exists a reasonable encoding $[\![\cdot]\!]$. Let a, b, c and d be pairwise distinct names; let Ω denote a divergent process and define $P \triangleq \textbf{if } x = d \textbf{ then } \Omega$. Faithfulness implies that $[\![\, \textbf{if } d = d \textbf{ then } \Omega \,]\!]$ diverges, $[\![\, a(x).P \,]\!]$ cannot offer data and $[\![\, \overline{a}\langle b\rangle \,]\!]$ must offer some datum. Moreover, because of Proposition 5, $[\![\, a(x).P \,]\!]$ and $[\![\, \overline{a}\langle b\rangle \,]\!]$ cannot perform τ-steps in isolation; however, because of operational correspondence, when put in parallel they must perform at least one τ-step to become $[\![\, P\{^b/_x\} \,]\!]$. If the input performed by $[\![\, a(x).P \,]\!]$ relies on a formal field, then we can obtain a divergent computation from $[\![\, a(x).P \mid \overline{c}\langle b\rangle \,]\!]$. So, it must be that $[\![\, a(x).P \,]\!]$ starts with an input relying on an actual template field $\ulcorner a\urcorner$ (it must be a otherwise, by name invariance, $[\![\, a(x).P \mid \overline{c}\langle b\rangle \,]\!]$ would diverge).

Now, it is easy to prove that $[\![\, a(x).P \,]\!] \mid [\![\, \overline{a}\langle b\rangle \,]\!] \Rightarrow [\![\, P\{^b/_x\} \,]\!]$ if and only if $[\![\, a(x).P \,]\!] \xRightarrow{\rho} R$, $[\![\, \overline{a}\langle b\rangle \,]\!] \xRightarrow{\overline{\rho}} R'$ and $R \mid R' \equiv [\![\, P\{^b/_x\} \,]\!]$. It must be that $\rho \triangleq ?a \cdot \rho_1 \cdot ?b \cdot \rho_2$, for b not occurring in ρ_1 and $?b$ generated by an input action with a formal template field; moreover, $[\![\, a(x).P \,]\!] \xRightarrow{?a\cdot\rho_1} R_1 \xrightarrow{?b} R_2 \xRightarrow{\rho_2} R$ and $[\![\, \overline{a}\langle b\rangle \,]\!] \xRightarrow{!a\cdot\overline{\rho_1}} R_3 \xrightarrow{!b} R_4 \xRightarrow{\overline{\rho_2}} R'$. In particular, $\rho_2 \triangleq \Box_1 n_1 \cdot \ldots \cdot \Box_k n_k$, for $\Box_i \in \{?,!\}$, and $\overline{\rho_2} \triangleq \Diamond_1 n_1 \cdot \ldots \cdot \Diamond_k n_k$, for $\Diamond_i \in \{?,!\} - \{\Box_i\}$.

Let σ be the permutation that swaps a and c and b and d; by name invariance, $[\![\, c(x).P \,]\!] \xRightarrow{?c\cdot\rho_1'} R_1\sigma \xrightarrow{?d} R_2\sigma \xRightarrow{\rho_2'} R\sigma$ and $[\![\, \overline{c}\langle d\rangle \,]\!] \xRightarrow{!c\cdot\overline{\rho_1'}} R_3\sigma \xrightarrow{!d} R_4\sigma \xRightarrow{\overline{\rho_2'}} R'\sigma$, for $\rho_1' = \rho_1\sigma$ and $\rho_2' = \rho_2\sigma$. More precisely, $\rho_2' \triangleq \Box_1 n_1' \cdot \ldots \cdot \Box_k n_k'$ and $\overline{\rho_2'} \triangleq \Diamond_1 n_1' \cdot \ldots \cdot \Diamond_k n_k'$, for $n_i' \triangleq \sigma(n_i)$.

Now, consider $Q \triangleq a(x).P | \overline{a}\langle b\rangle \mid \overline{c}\langle d\rangle \mid c(x).P'$, where $P' \triangleq \mathbf{if}\ x = b\ \mathbf{then}\ \Omega$; trivially, $Q \not\Uparrow$ while, as we shall see, $[\![Q]\!] \Uparrow$. This yields the desired contradiction. Consider

$$[\![Q]\!] \Rightarrow R_1 \mid R_3 \mid R_1\sigma \mid R_3\sigma \longrightarrow R_2\{d/b\} \mid R_4 \mid (R_2\sigma)\{b/d\} \mid R_4\sigma$$

where R_1 received d in place of b and $R_1\sigma$ received b in place of d (this is possible since these inputs do not rely on actual fields). Now, $R_2\{d/b\} \xRightarrow{\Box_1 m_1 \cdot \ldots \cdot \Box_k m_k}$, where

$$m_k \triangleq \begin{cases} d & \text{if } n_i = b \\ n_i & \text{otherwise} \end{cases}$$

and $(R_2\sigma)\{b/d\} \xRightarrow{\Box_1 m'_1 \cdot \ldots \cdot \Box_k m'_k}$, where

$$m'_k \triangleq \begin{cases} b & \text{if } n'_i = d \\ n'_i & \text{otherwise} \end{cases}$$

Finally, consider the computation

$$R_2\{d/b\} \mid R_4 \mid (R_2\sigma)\{b/d\} \mid R_4\sigma \Rightarrow R\{d/b\} \mid R'\{d/b\} \mid (R\sigma)\{b/d\} \mid (R'\sigma)\{b/d\}$$

obtained by performing a communication

- between $\Box_i m_i$ and $\Diamond_i n_i$ and between $\Box_i m'_i$ and $\Diamond_i n'_i$, if $n_i \neq b$, or
- between $\Box_i m_i$ and $\Diamond_i n'_i$ and between $\Box_i m'_i$ and $\Diamond_i n_i$, otherwise.

Now, $R\{d/b\} \mid R'\{d/b\} \triangleq (R \mid R')\{d/b\} \equiv [\![P\{b/x\}]\!]\{d/b\} \triangleq [\![\mathbf{if}\ d = d\ \mathbf{then}\ \Omega]\!]$, that is a divergent process. □

Theorem 6. *There exists no reasonable encoding of* π_{100} *in* π_{001}.

Proof. The proof is similar to that of Theorem 5. Assume that $[\![\cdot]\!]$ is reasonable; consider the process $P \triangleq (x, y).\mathbf{if}\ x = a\ \mathbf{then\ if}\ y = d\ \mathbf{then}\ \Omega$; pick up $c \neq a$ and $d \neq b$; consider the permutation of names σ swapping a with c and b with d; finally, show that $Q \triangleq P \mid \langle a, b\rangle \mid P\sigma \mid \langle c, d\rangle$ is not divergent, while $[\![Q]\!] \Uparrow$. □

6 Conclusion and Related Work

We have studied the expressive power of eight communication primitives, arising from the combination of three features: arity of data, communication medium and presence of pattern-matching. By relying on possibility/impossibility of 'reasonable' encodings, we obtained a clear hierarchy of communication primitives. Notably, LINDA's communication paradigm [15] is at the top of this hierarchy, while the π-calculus is in the middle. A posteriori, this can justify the fact that the former one is usually exploited in actual programming languages [3, 14], where flexibility and expressive power are the driving issues, while the latter one is mostly used for theoretical reasoning.

One of the pioneering works in the study of communication primitives for distributed systems is [16]. There, the expressive power of several "classical" primitives

(like test-and-set, compare-and-swap, ...) is studied by associating to every primitive the highest number of parallel processes that can reach a distributed consensus with that primitive, under conditions similar to the 'reasonableness' of our Definition 2. It then follows that a primitive with number n is less expressive than every primitive with number m ($> n$): the latter one can solve a problem (i.e. the consensus among m processes) that the former one cannot reasonably solve. This idea is also exploited in [22] to assess the expressive power of the non-deterministic choice in the π-calculus.

In [10], the notion of relative expressive power is used to measure the expressiveness of programming languages. In particular, a simple class of three concurrent constraint languages is studied and organised in a strict hierarchy. The languages have guarded constructs and only differ in the features offered by the guards: a guard is always passed in the less expressive language; a guard is passed only if a given constraint is satisfied by the current knowledge; and, finally, a guard is passed only if a new constraint, that must be atomically added to the knowledge, is consistent with the current knowledge. Very roughly, the last kind of guards can be related to the pattern-matching construct of our calculi, for the possibility of atomically testing and modifying the environment; in both cases, this feature sensibly increases the expressive power of the language.

By the way, the form of pattern-matching considered here is very minimal: only the equality of names can be tested while retrieving a datum. However, many other forms of pattern-matching can be exploited [12], to yield more and more flexible formalisms; some proposals have been investigated from the expressiveness point of view in [28].

Finally, in [7] a form of atomic polyadic name matching is presented, but with a different approach w.r.t. ours. Indeed, while in our π_{111} the tuple of names to be matched is in the transmitted/received value (by using a standard π-calculus terminology, the tuple is in the 'object' part of an output/input), in [7] there are composite channel names that must be matched to enable a communication (thus, the tuple is in the 'subject' part of the output/input). This feature enables a nice modelling of distributed and cryptographic process calculi; nevertheless, our LINDA-like pattern-matching is stronger, since the possibility of using both formal and actual fields in a template yield a more flexible form of input actions.

To conclude, this paper is one of the first attempts to classify languages according to their communication primitive. A lot of work still remains to be done. For example, it would be interesting to study more concrete languages, maybe by encoding them in one of the calculi presented in this paper. Moreover, other common features (such as synchrony) could be added to the picture. Finally, it would also be interesting to prove stronger properties for the encodings of Section 4, whenever possible; indeed, since we were mostly interested in the impossibility results, we intentionally exploited quite a weak form of 'reasonableness'.

Acknowledgements. I would like to thank Rosario Pugliese for his suggestions that improved a first draft of this paper and Catuscia Palamidessi for her encouragements and discussions. Finally, I also thank the anonymous *FoSSaCS'06* referees for their positive attitude and fruitful comments.

References

1. L. Acciai and M. Boreale. XPi: a typed process calculus for XML messaging. In *Proc. of FMOODS'05*, volume 3535 of *LNCS*, pages 47–66. Springer, 2005.
2. R. M. Amadio, I. Castellani and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.
3. K. Arnold, E. Freeman and S. Hupfer. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.
4. G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.
5. A. Brown, C. Laneve and G. Meredith. πduce: a process calculus with native XML datatypes. In *Proc. of Services and Formal Methods*, volume 3670 of *LNCS*. Springer, 2005.
6. N. Busi, R. Gorrieri and G. Zavattaro. A process algebraic view of LINDA coordination primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.
7. M. Carbone and S. Maffeis. On the expressive power of polyadic synchronisation in π-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.
8. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1): 177–213, 2000.
9. G. Castagna, R. De Nicola and D. Varacca. Semantic subtyping for the π-calculus. In *Proc. of LICS'05*, pages 92–101. IEEE Computer Society, 2005.
10. F. de Boer and C. Palamidessi. Embedding as a tool for language comparison. *Information and Computation*, 108(1):128–157, 1994.
11. R. De Nicola, D. Gorla and R. Pugliese. On the expressive power of KLAIM-based calculi. Tech. Rep. 09/2004, Dip. Informatica, Univ. Roma "La Sapienza". To appear in *TCS*.
12. R. De Nicola, D. Gorla and R. Pugliese. Pattern Matching over a Dynamic Network of Tuple Spaces. In *Proc. of FMOODS'05*, volume 3535 of *LNCS*, pages 1–14. Springer, 2005.
13. C. Ene and T. Muntean. Expressiveness of point-to-point versus broadcast communications. In *Proc. of FCT'99*, volume 1684 of *LNCS*, pages 258–268. Springer, 1999.
14. D. Ford, T. Lehman, S. McLaughry and P. Wyckoff. T Spaces. *IBM Systems Journal*, pages 454–474, August 1998.
15. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
16. M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. of PODC'88*, pages 276–290. ACM Press, 1988.
17. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP'91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.
18. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
19. R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.
20. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. of ICALP'92*, volume 623 of *LNCS*, pages 685–695. Springer, 1992.
21. U. Nestmann and B. Pierce. Decoding choice encodings. *Information and Computation*, 163:1–59, 2000.
22. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.
23. J. Parrow. An introduction to the π-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.
24. P. Quaglia and D. Walker. On encoding pπ in mπ. In *Proc. of FSTTCS'98*, volume 1530 of *LNCS*, pages 42–51. Springer, 1998.

25. D. Sangiorgi. Bisimulation in higher-order process calculi. *Information and Computation*, 131:141–178, 1996.
26. E. Shapiro. Separating concurrent languages with categories of language embeddings. In *Proc. of 23rd STOC*, pages 198–208. ACM Press, 1991.
27. N. Yoshida. Graph types for monadic mobile processes. In *Proc. of FSTTCS'96*, volume 1180 of *LNCS*, pages 371–386. Springer, 1996.
28. G. Zavattaro. Towards a hierarchy of negative test operators for generative communication. In *Proc. of EXPRESS'98*, volume 16 of *ENTCS*, 1998.

More on Bisimulations for Higher Order π-Calculus*

Zining Cao

Department of Computer Science and Engineering,
Nanjing University of Aero. & Astro., Nanjing 210016, P.R. China
caozn@nuaa.edu.cn

Abstract. In this paper, we prove the coincidence between strong/weak context bisimulation and strong/weak normal bisimulation for higher order π-calculus, which generalizes Sangiorgi's work. To achieve this aim, we introduce indexed higher order π-calculus, which is similar to higher order π-calculus except that every prefix of any process is assigned to indices. Furthermore we present corresponding indexed bisimulations for this calculus, and prove the equivalence between these indexed bisimulations. As an application of this result, we prove the equivalence between strong/weak context bisimulation and strong/weak normal bisimulation.

1 Introduction

Higher order π-calculus was proposed and studied intensively in Sangiorgi's dissertation [6]. It is an extension of the π-calculus [5] to allow communication of processes rather than names alone. In [6], some interesting bisimulations for higher order π-calculus were presented, such as barbed equivalence, context bisimulation and normal bisimulation. Barbed equivalence can be regarded as a uniform definition of bisimulation for a variety of concurrency calculi. Context bisimulation is a very intuitive definition of bisimulation for higher order π-calculus, but it is heavy to handle, due to the appearance of universal quantifications in its definition. In the definition of normal bisimulation, all universal quantifications disappeared, therefore normal bisimulation is a very economic characterisation of bisimulation for higher order π-calculus.

The main difficulty with definitions of context bisimulation and barbed equivalence that involve quantification over contexts is that they are often awkward to work with directly. It is therefore important to look for more tractable characterisations of the bisimulations. In [6, 7], the equivalence between weak normal bisimulation, weak context bisimulation and weak barbed equivalence was proved for early and late semantics respectively, but the proof method cannot be adapted to prove the equivalence between strong context bisimulation and strong normal bisimulation.

To the best of our knowledge, no paper gives the proof of equivalence between strong context bisimulation and strong normal bisimulation. In [7], this

* This work was supported by the National Science Foundation of China under Grant 60473036.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 63–78, 2006.
© Springer-Verlag Berlin Heidelberg 2006

problem was stated as an open problem. The main difficulty is that the proof strategy for the equivalence between weak context bisimulation and weak normal bisimulation does not work for the strong case. Roughly speaking, for the case of weak bisimulations, the mapping to triggered processes will bring some redundant tau actions. Since weak bisimulations abstract from tau action, the problem is inessential. But for the case of strong bisimulations, the situation is different. We have to match these redundant tau actions to prove that two processes are bisimilar. Therefore we need some new proof strategies to solve the problem.

The main aim of this paper is to give a uniform proof for the equivalence between strong/weak context bisimulation and strong/weak normal bisimulation. Especially, we will give a proof of the coincidence between strong context bisimulation and strong normal bisimulation, which solves an open problem presented by Sangiorgi in [7]. To achieve this aim, we introduce the notion of indexed processes and define several bisimulations on indexed processes such as indexed context bisimulation and indexed normal bisimulation. Furthermore, we present indexed triggered mapping, prove an indexed factorisation theorem, and give the equivalence between these indexed bisimulations. As an application of this result, we get a uniform proof for the equivalence between strong/weak context bisimulation and strong/weak normal bisimulation.

This paper is organized as follows: Section 2 gives a brief review of syntax and operational semantics of the higher order π-calculus, then recalls the definitions of context and normal bisimulations. Section 3 introduces indexed higher order π-calculus and some indexed bisimulations. The equivalence between these indexed bisimulations also be proved. In Section 4 we give a proof for the equivalence between strong/weak context bisimulation and strong/weak normal bisimulation. The paper is concluded in section 5.

2 Higher Order π-Calculus

2.1 Syntax and Labelled Transition System of Higher Order π-Calculus

In this section we briefly recall the syntax and labelled transition system of the higher order π-calculus. Similar to [7], we only focus on a second-order fragment of the higher order π-calculus, i.e., there is no abstraction in this fragment.

We assume a set N of names, ranged over by $a, b, c, ...$ and a set Var of process variables, ranged over by $X, Y, Z, U, ...$ We use $E, F, P, Q, ...$ to stand for processes. Pr denotes the set of all processes.

We first give the grammar for the higher order π-calculus processes as follows:

$P ::= 0 \mid U \mid \pi.P \mid P_1|P_2 \mid (\nu a)P \mid !P$

π is called a prefix and can have one of the following forms:

$\pi ::= \tau \mid l \mid \overline{l} \mid a(U) \mid \overline{a}\langle P\rangle$, here τ is a tau prefix; l is a first order input prefix; $\overline{l}$ is a first order output prefix; $a(U)$ is a higher order input prefix and $\overline{a}\langle P\rangle$ is a higher order output prefix.

Table 1.

$$ALP : \frac{P \xrightarrow{\alpha} P'}{Q \xrightarrow{\alpha} Q'} P \equiv_\alpha Q, P' \equiv_\alpha Q' \qquad TAU : \tau.P \xrightarrow{\tau} P$$

$$OUT1 : \overline{l}.P \xrightarrow{\overline{l}} P \qquad IN1 : l.P \xrightarrow{l} P$$

$$OUT2 : \overline{a}\langle E\rangle.P \xrightarrow{\overline{a}\langle E\rangle} P \qquad IN2 : a(U).P \xrightarrow{a\langle E\rangle} P\{E/U\}$$

$$PAR : \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} bn(\alpha) \cap fn(Q) = \emptyset$$

$$COM1 : \frac{P \xrightarrow{\overline{l}} P' \quad Q \xrightarrow{l} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$COM2 : \frac{P \xrightarrow{(\nu\widetilde{b})\overline{a}\langle E\rangle} P' \quad Q \xrightarrow{a\langle E\rangle} Q'}{P|Q \xrightarrow{\tau} (\nu\widetilde{b})(P'|Q')} \widetilde{b} \cap fn(Q) = \emptyset$$

$$RES : \frac{P \xrightarrow{\alpha} P'}{(\nu a)P \xrightarrow{\alpha} (\nu a)P'} a \notin n(\alpha) \qquad REP : \frac{P|!P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

$$OPEN : \frac{P \xrightarrow{(\nu\widetilde{c})\overline{a}\langle E\rangle} P'}{(\nu b)P \xrightarrow{(\nu b,\widetilde{c})\overline{a}\langle E\rangle} P'} a \neq b,\ b \in fn(E) - \widetilde{c}$$

For higher order π-calculus, the notations of free name, bound name, free variable, bound variable and etc are given in [6, 7]. The set of all closed processes, i.e., the processes which have no free variable, is denoted as Pr^c.

The operational semantics of higher order processes is given in Table 1. We have omitted the symmetric of the parallelism and communication rules.

2.2 Bisimulations in Higher Order π-Calculus

Context and normal bisimulations were presented in [6, 7] to describe the behavioral equivalences for higher order π-calculus. In the following, we abbreviate $P\{E/U\}$ as $P\langle E\rangle$.

Definition 1. A symmetric relation $R \subseteq Pr^c \times Pr^c$ is a strong context bisimulation if $P\ R\ Q$ implies:

(1) whenever $P \xrightarrow{\alpha} P'$, there exists Q' such that $Q \xrightarrow{\alpha} Q'$, here α is not a higher order action and $P'\ R\ Q'$;

(2) whenever $P \xrightarrow{a\langle E\rangle} P'$, there exists Q' such that $Q \xrightarrow{a\langle E\rangle} Q'$ and $P'\ R\ Q'$;

(3) whenever $P \xrightarrow{(\nu\widetilde{b})\overline{a}\langle E\rangle} P'$, there exist Q', F, $\widetilde{c}$ such that $Q \xrightarrow{(\nu\widetilde{c})\overline{a}\langle F\rangle} Q'$ and for all $C(U)$ with $fn(C(U)) \cap \{\widetilde{b}, \widetilde{c}\} = \emptyset$, $(\nu\widetilde{b})(P'|C\langle E\rangle)\ R\ (\nu\widetilde{c})(Q'|C\langle F\rangle)$. Here $C(U)$ is a process containing a unique variable U.

We write $P \sim_{Ct} Q$ if P and Q are strong context bisimilar.

Distinguished from strong context bisimulation, strong normal bisimulation does not have universal quantifications in the clauses of its definition. In the following, a name is called fresh in a statement if it is different from any other name occurring in the processes of the statement.

Definition 2. A symmetric relation $R \subseteq Pr^c \times Pr^c$ is a strong normal bisimulation if $P\ R\ Q$ implies:

(1) whenever $P \xrightarrow{\alpha} P'$, there exists Q' such that $Q \xrightarrow{\alpha} Q'$, here α is not a higher order action and $P'\ R\ Q'$;
(2) whenever $P \stackrel{a\langle \overline{m}.0\rangle}{\longrightarrow} P'$, there exists Q' such that $Q \stackrel{a\langle \overline{m}.0\rangle}{\longrightarrow} Q'$ and $P'\ R\ Q'$, here m is a fresh name;
(3) whenever $P \stackrel{(\nu\widetilde{b})\overline{a}\langle E\rangle}{\longrightarrow} P'$, there exist Q', F, $\widetilde{c}$ such that $Q \stackrel{(\nu\widetilde{c})\overline{a}\langle F\rangle}{\longrightarrow} Q'$ and $(\nu\widetilde{b})(P'|!m.E)\ R\ (\nu\widetilde{c})(Q'|!m.F)$, here m is a fresh name.

We write $P \sim_{Nr} Q$ if P and Q are strong normal bisimilar.

In the following, we use $\stackrel{\varepsilon}{\Longrightarrow}$ to abbreviate the reflexive and transitive closure of $\stackrel{\tau}{\longrightarrow}$, and use $\stackrel{\alpha}{\Longrightarrow}$ to abbreviate $\stackrel{\varepsilon}{\Longrightarrow}\stackrel{\alpha}{\longrightarrow}\stackrel{\varepsilon}{\Longrightarrow}$. By neglecting the tau action, we can get the following formal definitions of weak bisimulations:

Definition 3. A symmetric relation $R \subseteq Pr^c \times Pr^c$ is a weak context bisimulation if $P\ R\ Q$ implies:

(1) whenever $P \stackrel{\varepsilon}{\Longrightarrow} P'$, there exists Q' such that $Q \stackrel{\varepsilon}{\Longrightarrow} Q'$ and $P'\ R\ Q'$;
(2) whenever $P \stackrel{\alpha}{\Longrightarrow} P'$, there exists Q' such that $Q \stackrel{\alpha}{\Longrightarrow} Q'$, here α is not a higher order action, $\alpha \neq \tau$ and $P'\ R\ Q'$;
(3) whenever $P \stackrel{a\langle E\rangle}{\Longrightarrow} P'$, there exists Q' such that $Q \stackrel{a\langle E\rangle}{\Longrightarrow} Q'$ and $P'\ R\ Q'$;
(4) whenever $P \stackrel{(\nu\widetilde{b})\overline{a}\langle E\rangle}{\Longrightarrow} P'$, there exist Q', F, $\widetilde{c}$ such that $Q \stackrel{(\nu\widetilde{c})\overline{a}\langle F\rangle}{\Longrightarrow} Q'$ and for all $C(U)$ with $fn(C(U)) \cap \{\widetilde{b}, \widetilde{c}\} = \emptyset$, $(\nu\widetilde{b})(P'|C\langle E\rangle)\ R\ (\nu\widetilde{c})(Q'|C\langle F\rangle)$.

We write $P \approx_{Ct} Q$ if P and Q are weak context bisimilar.

Definition 4. A symmetric relation $R \subseteq Pr^c \times Pr^c$ is a weak normal bisimulation if $P\ R\ Q$ implies:

(1) whenever $P \stackrel{\varepsilon}{\Longrightarrow} P'$, there exists Q' such that $Q \stackrel{\varepsilon}{\Longrightarrow} Q'$ and $P'\ R\ Q'$;
(2) whenever $P \stackrel{\alpha}{\Longrightarrow} P'$, there exists Q' such that $Q \stackrel{\alpha}{\Longrightarrow} Q'$, here α is not a higher order action, $\alpha \neq \tau$ and $P'\ R\ Q'$;
(3) whenever $P \stackrel{a\langle \overline{m}.0\rangle}{\Longrightarrow} P'$, there exists Q' such that $Q \stackrel{a\langle \overline{m}.0\rangle}{\Longrightarrow} Q'$ and $P'\ R\ Q'$, here m is a fresh name;
(4) whenever $P \stackrel{(\nu\widetilde{b})\overline{a}\langle E\rangle}{\Longrightarrow} P'$, there exist Q', F, $\widetilde{c}$ such that $Q \stackrel{(\nu\widetilde{c})\overline{a}\langle F\rangle}{\Longrightarrow} Q'$ and $(\nu\widetilde{b})(P'|!m.E)\ R\ (\nu\widetilde{c})(Q'|!m.F)$, here m is a fresh name.

We write $P \approx_{Nr} Q$ if P and Q are weak normal bisimilar.

3 Indexed Processes and Indexed Bisimulations

3.1 Syntax and Labelled Transition System of Indexed Higher Order π-Calculus

The aim of this paper is to propose a general argument for showing the correspondence of context and normal bisimulations in both the strong and weak cases, by relying on a notion of indexed processes. Roughly, the intention is that indexed processes allow the labelled transition system semantics to record in action labels the indices of the interacting components. This mechanism is then used to filter out some tau transitions in the considered definition of bisimulation.

Now we introduce the concept of indexed processes.The index set I, w.l.o.g., will be the set of natural numbers. Intuitively, the concept of index can be viewed as the name or location of components. The class of the indexed processes IPr is built similar to Pr, except that every prefix is assigned to indices. We usually use K, L, M, N to denote indexed processes.

The formal definition of indexed process is given as follows:

$M ::= 0 \mid U \mid I\pi.M \mid M_1|M_2 \mid (\nu a)M \mid !M$

$I\pi$ is called indexed prefix and can be an indexed tau prefix or an indexed input prefix or an indexed output prefix:

$I\pi ::= \{\tau\}_{i,j} \mid \{l\}_i \mid \{\overline{l}\}_i \mid \{a(U)\}_i \mid \{\overline{a}\langle N\rangle\}_i$, i, j $\in$index set I (here N is an indexed process).

Similar to the original higher order π-calculus, in each indexed process of the form $(\nu a)M$ the occurrence of a is bound within the scope of M. An occurrence of a in M is said to be free iff it does not lie within the scope of a bound occurrence of a. The set of names occurring free in M is denoted $fn(M)$. An occurrence of a name in M is said to be bound if it is not free, we write the set of bound names as $bn(M)$. $n(M)$ denotes the set of names of M, i.e., $n(M) = fn(M) \cup bn(M)$. We use $n(M,N)$ to denote $n(M) \cup n(N)$. Indexed higher order input prefix $\{a(U)\}_i.M$ binds all free occurrences of U in M. The set of variables occurring free in M is denoted as $fv(M)$. We write the set of bound variables in M as $bv(M)$. An indexed process is closed if it has no free variable; it is open if it may have free variables. IPr^c is the set of all closed indexed processes. Indexed processes M and N are α-convertible, $M \equiv_\alpha N$, if N can be obtained from M by a finite number of changes of bound names and bound variables.

The set of all indices that occur in M, $Index(M)$, is defined inductively as follows:

(1) if $M = 0$ or U, then $Index(M) ::= \emptyset$;
(2) if $M = I\pi.M_1$, then $Index(M) ::= Index(I\pi) \cup Index(M_1)$, here $Index(I\pi)$ $::= \{i,j\}$ if $I\pi$ is in the form of $\{\tau\}_{i,j}$; $Index(I\pi) ::= \{i\} \cup Index(N)$ if $I\pi$ is in the form of $\{\overline{x}\langle N\rangle\}_i$; $Index(I\pi) ::= \{i\}$ if $I\pi$ is in the form of $\{l\}_i$ or $\{\overline{l}\}_i$ or $\{x(U)\}_i$.
(3) if $M = M_1|M_2$, then $Index(M) ::= Index(M_1) \cup Index(M_2)$;
(4) if $M = (\nu a)M_1$, then $Index(M) ::= Index(M_1)$;
(5) if $M = !M_1$, then $Index(M) ::= Index(M_1)$.

We use $Index(M,N)$ to denote $Index(M) \cup Index(N)$.

In the remainder of this paper, $\{P\}_i$ is an abbreviation for the indexed process with the same given index i on every prefix in the scope of P. The formal definition can be given inductively as follows:

(1) $\{0\}_i ::= 0$;
(2) $\{U\}_i ::= U$;
(3) $\{\tau.P\}_i ::= \{\tau\}_{i,i}.\{P\}_i$;
(4) $\{l.P\}_i ::= \{l\}_i.\{P\}_i$;
(5) $\{\overline{l}.P\}_i ::= \{\overline{l}\}_i.\{P\}_i$;
(6) $\{a(U).P\}_i ::= \{a(U)\}_i.\{P\}_i$;
(7) $\{\overline{a}\langle E\rangle.P\}_i ::= \{\overline{a}\langle\{E\}_i\rangle\}_i.\{P\}_i$;
(8) $\{P_1|P_2\}_i ::= \{P_1\}_i|\{P_2\}_i$;
(9) $\{(\nu a)P\}_i ::= (\nu a)\{P\}_i$;
(10) $\{!P\}_i ::= !\{P\}_i$.

In the labelled transition system of indexed higher order π-calculus, the label on the transition arrow is an indexed action, whose definition is given as follows:

$I\alpha ::= \{\tau\}_{i,j} \mid \{l\}_i \mid \{\overline{l}\}_i \mid \{a\langle K\rangle\}_i \mid \{\overline{a}\langle K\rangle\}_i \mid \{(\nu\widetilde{b})\overline{a}\langle K\rangle\}_i$, here $\{\tau\}_{i,j}$ is an indexed tau action, $\{l\}_i$ is an indexed first order input action, $\{\overline{l}\}_i$ is an indexed first order output action, $\{a\langle K\rangle\}_i$ is an indexed higher order input action, and $\{\overline{a}\langle K\rangle\}_i$ and $\{(\nu\widetilde{b})\overline{a}\langle K\rangle\}_i$ are indexed higher order output actions.

We write $bn(I\alpha)$ to represent the set of names bound in $I\alpha$, which is $\{\widetilde{b}\}$ if $I\alpha$ is $\{(\nu\widetilde{b})\overline{a}\langle K\rangle\}_i$ and $\emptyset$ otherwise. $n(I\alpha)$ is the set of names that occur in $I\alpha$.

Table 2.

$$ALP : \frac{M \xrightarrow{I\alpha} M'}{N \xrightarrow{I\alpha} N'} M \equiv_\alpha N, M' \equiv_\alpha N' \qquad TAU : \{\tau\}_{i,j}.M \xrightarrow{\{\tau\}_{i,j}} M$$

$$OUT1 : \{\overline{l}\}_i.M \xrightarrow{\{\overline{l}\}_i} M \qquad IN1 : \{l\}_i.M \xrightarrow{\{l\}_i} M$$

$$OUT2 : \{\overline{a}\langle K\rangle\}_i.M \xrightarrow{\{\overline{a}\langle K\rangle\}_i} M \qquad IN2 : \{a(U)\}_i.M \xrightarrow{\{a\langle K\rangle\}_i} M\{K/U\}$$

$$PAR : \frac{M \xrightarrow{I\alpha} M'}{M|N \xrightarrow{I\alpha} M'|N} bn(I\alpha) \cap fn(N) = \emptyset$$

$$COM1 : \frac{M \xrightarrow{\{\overline{l}\}_i} M' \quad N \xrightarrow{\{l\}_j} N'}{M|N \xrightarrow{\{\tau\}_{i,j}} (M'|N')}$$

$$COM2 : \frac{M \xrightarrow{\{(\nu\widetilde{b})\overline{a}\langle K\rangle\}_i} M' \quad N \xrightarrow{\{a\langle K\rangle\}_j} N'}{M|N \xrightarrow{\{\tau\}_{i,j}} (\nu\widetilde{b})(M'|N')} \widetilde{b} \cap fn(N) = \emptyset$$

$$RES : \frac{M \xrightarrow{I\alpha} M'}{(\nu a)M \xrightarrow{I\alpha} (\nu a)M'} a \notin n(I\alpha) \qquad REP : \frac{M|!M \xrightarrow{I\alpha} M'}{!M \xrightarrow{I\alpha} M'}$$

$$OPEN : \frac{M \xrightarrow{\{(\nu\widetilde{c})\overline{a}\langle K\rangle\}_i} M'}{(\nu b)M \xrightarrow{\{(\nu b,\widetilde{c})\overline{a}\langle K\rangle\}_i} M'} a \neq b, b \in fn(K) - \widetilde{c}$$

The operational semantics of indexed processes is given in Table 2. Similar to Table 1, we have omitted the symmetric of the parallelism and communication. The main difference between Table 1 and Table 2 is that the label $I\alpha$ on the transition arrow is in the form of $\{\alpha\}_i$ or $\{\tau\}_{i,j}$. If we adopt the distributed view, $\{\alpha\}_i$ can be regarded as an input or output action performed by component i, and $\{\tau\}_{i,j}$ can be regarded as a communication between components i and j.

Remark: Since $\{\tau\}_{i,j}$ and $\{\tau\}_{j,i}$ have the same meaning: a communication between components i and j, hence i, j should be considered as a set $\{i, j\}$, and not as an ordered pair. Therefore in the above labelled transition system, $\{\tau\}_{i,j}$ and $\{\tau\}_{j,i}$ are considered as the same label, i.e., $M \stackrel{\{\tau\}_{i,j}}{\longrightarrow} M'$ is viewed to be same as $M \stackrel{\{\tau\}_{j,i}}{\longrightarrow} M'$.

3.2 Indexed Context Bisimulation and Indexed Normal Bisimulation

Now we can give the concept of indexed context bisimulation and indexed normal bisimulation for indexed processes. In the remainder of this paper, we abbreviate $M\{K/U\}$ as $M\langle K\rangle$. In the following, we use $M \stackrel{\varepsilon,S}{\Longrightarrow} M'$ to abbreviate $M \stackrel{\{\tau\}_{i_1,i_1}}{\longrightarrow} ... \stackrel{\{\tau\}_{i_n,i_n}}{\longrightarrow} M'$, and use $M \stackrel{I\alpha,S}{\Longrightarrow} M'$ to abbreviate $M \stackrel{\varepsilon,S}{\Longrightarrow}\stackrel{I\alpha}{\longrightarrow}\stackrel{\varepsilon,S}{\Longrightarrow} M'$, here $i_1, ..., i_n \in S \subseteq I$. An index is called fresh in a statement if it is different from any other index occurring in the processes of the statement. Let us see two examples. For the transition $(\nu a)((\nu b)(\{\overline{a}\}_n.0|\{\overline{b}\}_m.0|\{a\}_n.\{b\}_m.0)) \stackrel{\{\tau\}_{n,n}}{\longrightarrow}\stackrel{\{\tau\}_{m,m}}{\longrightarrow} 0$, we can abbreviate it as $(\nu a)((\nu b)(\{\overline{a}\}_n.0|\{\overline{b}\}_m.0|\{a\}_n.\{b\}_m.0)) \stackrel{\varepsilon,\{m,n\}}{\Longrightarrow} 0$. Similarly, since $(\nu a)((\nu b)(\{\overline{a}\}_n.0|\{\overline{b}\}_m.0|\{a\}_n.\{c\}_k.\{b\}_m.0)) \stackrel{\{\tau\}_{n,n}}{\longrightarrow}\stackrel{\{c\}_k}{\longrightarrow}\stackrel{\{\tau\}_{m,m}}{\longrightarrow} 0$, we can abbreviate it as $(\nu a)((\nu b)(\{\overline{a}\}_n.0|\{\overline{b}\}_m.0|\{a\}_n.\{c\}_k.\{b\}_m.0)) \stackrel{\{c\}_k,\{m,n\}}{\Longrightarrow} 0$.

This paper's main result states that strong context bisimulation coincides with strong normal bisimulation. Technically, the proof rests on the notion of indexed bisimulations. The idea is to generalize the usual notion of weak bisimulations so that tau actions can be ignored selectively, depending on a chosen set of indices S. The cases $S = \emptyset$ and $S = I$ correspond to strong and weak bisimulations respectively.

Definition 5. Let M, N be two closed indexed processes, and $S \subseteq I$ be an index set, we write $M \simeq^S_{Ct} N$, if there is a symmetric relation R and $M\ R\ N$ implies:

(1) whenever $M \stackrel{\varepsilon,S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{\varepsilon,S}{\Longrightarrow} N'$ and $M'\ R\ N'$;
(2) whenever $M \stackrel{I\alpha,S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{I\alpha,S}{\Longrightarrow} N'$ and $M'\ R\ N'$, here $I\alpha \neq \{\tau\}_{i,i}$ for any $i \in S$, $I\alpha$ is not an indexed higher order action;
(3) whenever $M \stackrel{\{a\langle K\rangle\}_i,S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{\{a\langle K\rangle\}_i,S}{\Longrightarrow} N'$ and $M'\ R\ N'$;

(4) whenever $M \stackrel{\{(\nu\widetilde{b})\overline{a}\langle K\rangle\}_{i},S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{\{(\nu\widetilde{c})\overline{a}\langle L\rangle\}_{i},S}{\Longrightarrow} N'$ and for any indexed process $C(U)$ with $fn(C(U))\cap\{\widetilde{b},\widetilde{c}\} = \oslash$, $(\nu\widetilde{b})(M'|C\langle K\rangle)$ R $(\nu\widetilde{c})(N'|C\langle L\rangle)$.

We say that M and N are indexed context bisimilar w.r.t. S if $M \simeq^{S}_{Ct} N$.

Definition 6. Let M, N be two closed indexed processes, and $S \subseteq I$ be an index set, we write $M \simeq^{S}_{Nr} N$, if there is a symmetric relation R and $M\ R\ N$ implies:

(1) whenever $M \stackrel{\varepsilon,S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{\varepsilon,S}{\Longrightarrow} N'$ and $M'\ R\ N'$;
(2) whenever $M \stackrel{I\alpha,S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{I\alpha,S}{\Longrightarrow} N'$ and $M'\ R\ N'$, here $I\alpha \neq \{\tau\}_{i,i}$ for any $i \in S$, $I\alpha$ is not an indexed higher order action;
(3) whenever $M \stackrel{\{a\langle\{\overline{m}\}_{n}.0\rangle\}_{i},S}{\Longrightarrow} M'$, here m is a fresh name, there exists N' such that $N \stackrel{\{a\langle\{\overline{m}\}_{n}.0\rangle\}_{i},S}{\Longrightarrow} N'$ and $M'\ R\ N'$;
(4) whenever $M \stackrel{\{(\nu\widetilde{b})\overline{a}\langle K\rangle\}_{i},S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{\{(\nu\widetilde{c})\overline{a}\langle L\rangle\}_{i},S}{\Longrightarrow} N'$, and $(\nu\widetilde{b})(M'|!\{m\}_{n}.K)\ R\ (\nu\widetilde{c})(N'|!\{m\}_{n}.L)$ with a fresh name m and a fresh index n.

We say that M and N are indexed normal bisimilar w.r.t. S if $M \simeq^{S}_{Nr} N$.

The above definitions have some geometric intuition. From a distributed view, $\{\tau\}_{i,i}$ is an internal communication in component i, and $\{\tau\}_{i,j}$, where $i \neq j$, represents an external communication between components i and j. Therefore in Definitions 5 and 6, we regard $\{\tau\}_{i,i}$ as a private event in component i, which can be neglected if i is in S, a chosen set of indices; and we view $\{\tau\}_{i,j}$ as a visible event between components i and j.

For example, by the above definition, we have $(\nu a)(\{\overline{a}\}_{n}.0|\{a\}_{n}.M) \simeq^{\{n\}}_{Ct} M$, $(\nu a)(\{\overline{a}\}_{n}.0|\{a\}_{n}.M) \not\simeq^{\emptyset}_{Ct} M$ and $(\nu a)(\{\overline{a}\}_{n}.0|\{a\}_{n}.M) \simeq^{I}_{Nr} M$.

3.3 Indexed Triggered Processes and Indexed Triggered Bisimulation

The concept of triggered processes was introduced in [6, 7]. The distinguishing feature of triggered processes is that every communication among them is the exchange of a trigger, here a trigger is an elementary process whose only functionality is to activate a copy of another process. In this section, we introduce the indexed version of triggered processes. Indexed triggered process can be seen as a sort of normal form for the indexed processes, and every communication among them is the exchange of an indexed trigger. We shall use indexed triggers to perform indexed process transformations which make the treatment of the constructs of indexed higher order processes easier.

The formal definition of indexed triggered process is given as follows:

$M ::= 0 \mid U \mid \{\tau\}_{i,j}.M \mid \{l\}_{i}.M \mid \{\overline{l}\}_{i}.M \mid \{a(U)\}_{i}.M \mid (\nu m)(\{\overline{a}\langle\{\overline{m}\}_{n}.0\rangle\}_{i}.M|$ $!\{m\}_{n}.N)$ with $m \notin fn(M,N) \cup \{a\} \mid M_1|M_2 \mid (\nu a)M \mid !M$.

The class of the indexed triggered processes is denoted as $ITPr$. The class of the closed indexed triggered processes is denoted as $ITPr^c$.

Definition 7. We give a mapping Tr^n which transforms every indexed process M into the indexed triggered process $Tr^n[M]$ with respect to index n. The mapping is defined inductively on the structure of M.

(1) $Tr^n[0] ::= 0$;
(2) $Tr^n[U] ::= U$;
(3) $Tr^n[\{\tau\}_{i,j}.M] ::= \{\tau\}_{i,j}.Tr^n[M]$;
(4) $Tr^n[\{l\}_i.M] ::= \{l\}_i.Tr^n[M]$;
(5) $Tr^n[\{\bar{l}\}_i.M] ::= \{\bar{l}\}_i.Tr^n[M]$;
(6) $Tr^n[\{a(U)\}_i.M] ::= \{a(U)\}_i.Tr^n[M]$;
(7) $Tr^n[\{\overline{a}\langle N\rangle\}_i.M] ::= (\nu m)(\{\overline{a}\langle\{\overline{m}\}_n.0\rangle\}_i.Tr^n[M]|!\{m\}_n.Tr^n[N])$, where m is a fresh name;
(8) $Tr^n[M_1|M_2] ::= Tr^n[M_1]|Tr^n[M_2]$;
(9) $Tr^n[(\nu a)M] ::= (\nu a)Tr^n[M]$;
(10) $Tr^n[!M] ::= !Tr^n[M]$.

Transformation $Tr^n[\ ]$ may expand the number of $\{\tau\}_{n,n}$ steps in a process. But the behavior is otherwise the same. The expansion is due to the fact that if in M a process N is transmitted and used k times then, in $Tr^n[M]$ k additional $\{\tau\}_{n,n}$ interactions are required to activate the copies of N.

For example, let $M \stackrel{def}{=} \{\overline{a}\langle N\rangle\}_i.L|\{a(U)\}_j.(U|U)$, then $M \stackrel{\{\tau\}_{i,j}}{\longrightarrow} L|N|N \stackrel{def}{=} M'$. In $Tr^n[M]$, this is simulated using two additional $\{\tau\}_{n,n}$ interactions:

$$\begin{aligned} Tr^n[M] &= (\nu m)(\{\overline{a}\langle\{\overline{m}\}_n.0\rangle\}_i.Tr^n[L]|!\{m\}_n.Tr^n[N])|\{a(U)\}_j.(U|U) \\ &\stackrel{\{\tau\}_{i,j}}{\longrightarrow} (\nu m)(Tr^n[L]|!\{m\}_n.Tr^n[N]|\{\overline{m}\}_n.0|\{\overline{m}\}_n.0) \\ &\stackrel{\{\tau\}_{n,n}}{\longrightarrow}\stackrel{\{\tau\}_{n,n}}{\longrightarrow} (\nu m)(Tr^n[L]|Tr^n[N]|Tr^n[N]|!\{m\}_n.Tr^n[N]) \\ &\simeq^{\emptyset}_{Ct} Tr^n[L]|Tr^n[N]|Tr^n[N] \text{ since } m \text{ is a fresh name} \\ &= Tr^n[M']. \end{aligned}$$

Now we can give the indexed version of triggered bisimulation as follows.

Definition 8. Let M, N be two closed indexed triggered processes, and $S \subseteq I$ be an index set, we write $M \simeq^S_{Tr} N$, if there is a symmetric relation R and M R N implies:

(1) whenever $M \stackrel{\varepsilon,S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{\varepsilon,S}{\Longrightarrow} N'$ and M' R N';
(2) whenever $M \stackrel{I\alpha,S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{I\alpha,S}{\Longrightarrow} N'$ and M' R N', here $I\alpha \neq \{\tau\}_{i,i}$ for any $i \in S$, $I\alpha$ is not an indexed higher order action;
(3) whenever $M \stackrel{\{a\langle\{\overline{m}\}_n.0\rangle\}_i,S}{\Longrightarrow} M'$, here m is a fresh name, there exists N' such that $N \stackrel{\{a\langle\{\overline{m}\}_n.0\rangle\}_i,S}{\Longrightarrow} N'$ and M' R N';
(4) whenever $M \stackrel{\{(\nu m)\overline{a}\langle\{\overline{m}\}_n.0\rangle\}_i,S}{\Longrightarrow} M'$, there exists N' such that $N \stackrel{\{(\nu m)\overline{a}\langle\{\overline{m}\}_n.0\rangle\}_i,S}{\Longrightarrow} N'$ and M' R N'.

We say that M and N are indexed triggered bisimilar w.r.t. S if $M \simeq^S_{Tr} N$.

3.4 The Equivalence Between Indexed Bisimulations

In [6, 7], the equivalence between weak context bisimulation and weak normal bisimulation was proved. In the proof, the factorisation theorem was firstly given. It allows us to factorise out certain subprocesses of a given process. Thus, a complex process can be decomposed into the parallel composition of simpler processes. Then the concept of triggered processes was introduced, which is the key step in the proof. Triggered processes represent a sort of normal form for the processes. Most importantly, there is a very simple characterisation of context bisimulation on triggered processes, called triggered bisimulation. By the factorisation theorem, a process can be transformed to a triggered process. The transform allows us to use the simpler theory of triggered processes to reason about the set of all processes. In [6, 7], weak context bisimulation was firstly proved to be equivalent to weak triggered bisimulation on triggered processes, then by the mapping from general processes to triggered processes, the equivalence between weak context bisimulation and weak normal bisimulation was proved.

In the case of strong bisimulations, the above proof strategy does not work. The main problem is that the mapping to triggered processes brings some redundant tau actions. Since weak bisimulations abstract from tau action, the full abstraction of the mapping to triggered processes holds. But in the case of strong bisimulations, the triggered mapping does not preserve the strong bisimulations, and therefore some central technical results in [6, 7], like the factorisation theorem, are not true in the strong case.

To resolve this difficulty we introduced the concept of indexed processes and the indexed version of context and normal bisimulations. Roughly, the actions of indexed processes have added indices, which are used to identify in which component or between which components an action takes place. Indexed bisimulations with respect to an indices set S neglect the indexed tau action of the form $\{\tau\}_{i,i}$ for any $i \in S$, but distinguish the indexed tau action of the form $\{\tau\}_{i,j}$ if $i \notin S$ or $j \notin S$ or $i \neq j$. One can see that the mapping from M to $Tr^n[M]$ brings redundant indexed tau actions $\{\tau\}_{n,n}$. Therefore indexed triggered mapping preserves the indexed bisimulations with respect to $S \cup \{n\}$ for any S. Similarly, we also have the indexed version of factorisation theorem. Following the proof strategy in [6, 7], we prove the equivalence between indexed context bisimulation and indexed normal bisimulation. Furthermore, when S is the empty set $\emptyset$, we discuss the relation between indexed bisimulations and strong bisimulations, and get the proposition: $P \sim_{Nr} Q \Leftrightarrow \{P\}_k \simeq^{\emptyset}_{Nr} \{Q\}_k \Leftrightarrow Tr^n[\{P\}_k] \simeq^{\{n\}}_{Tr} Tr^n[\{Q\}_k] \Leftrightarrow \{P\}_k \simeq^{\emptyset}_{Ct} \{Q\}_k \Leftrightarrow P \sim_{Ct} Q$. This solves the open problem in [7]. We also apply the proof idea to the case of weak bisimulations. When S is the full index set I, we study the relation between indexed bisimulations and weak bisimulations, and get the proposition: $P \approx_{Nr} Q \Leftrightarrow \{P\}_k \simeq^{I}_{Nr} \{Q\}_k \Leftrightarrow Tr^n[\{P\}_k] \simeq^{I}_{Tr} Tr^n[\{Q\}_k] \Leftrightarrow \{P\}_k \simeq^{I}_{Ct} \{Q\}_k \Leftrightarrow P \approx_{Ct} Q$. Therefore the proof presented here seems to be a uniform approach to the equivalence between strong/weak context bisimulation and strong/weak normal bisimulation.

Now we study the relations between the three indexed bisimulations. The main result is summarized in Proposition 8: $M \simeq^{S}_{Nr} N \Leftrightarrow Tr^n[M] \simeq^{S\cup\{n\}}_{Tr}$

$Tr^n[N] \Leftrightarrow M \simeq^S_{Ct} N$. We achieve this result by proving several propositions: including indexed factorisation theorem (Proposition 4), full abstraction of the mapping to indexed triggered processes (Proposition 5), the relation between indexed triggered bisimulation and indexed normal bisimulation (Proposition 6), and the relation between indexed triggered bisimulation and indexed context bisimulation (Proposition 7).

In the following, we first give congruence of $\simeq^S_{Ct}$ and $\simeq^S_{Tr}$.

Proposition 1 (Congruence of $\simeq^S_{Ct}$)**.** For all M, N, $K \in IPr^c$, $M \simeq^S_{Ct} N$ implies:

1. $I\pi.M \simeq^S_{Ct} I\pi.N$;
2. $M|K \simeq^S_{Ct} N|K$;
3. $(\nu a)M \simeq^S_{Ct} (\nu a)N$;
4. $!M \simeq^S_{Ct} !N$;
5. $\overline{a}\langle M\rangle.K \simeq^S_{Ct} \overline{a}\langle N\rangle.K$.

Proof : Similar to the argument of the analogous result for context bisimulation in [6, Theorem 4.2.7].

Proposition 2 (Congruence of $\simeq^S_{Tr}$)**.** For all M, N, $K \in ITPr^c$, $M \simeq^S_{Tr} N$ implies:

1. $M|K \simeq^S_{Tr} N|K$;
2. $(\nu a)M \simeq^S_{Tr} (\nu a)N$.

Proof : Similar to the argument of the analogous result for triggered bisimulation in [6, Lemma 4.6.3].

Proposition 3 states the easy part of the relation between $\simeq^S_{Ct}$ and $\simeq^S_{Nr}$.

Proposition 3. For any M, $N \in IPr^c$, $M \simeq^S_{Ct} N \Rightarrow M \simeq^S_{Nr} N$.

Proof : It is trivial by the definition of $\simeq^S_{Ct}$ and $\simeq^S_{Nr}$.

Now we give the indexed version of the factorisation theorem, which states that, by means of indexed triggers, an indexed subprocess of a given indexed process can be factorised out.

Proposition 4. For any indexed processes M and N with $m \notin fn(M, N)$, it holds that $M\{\{\tau\}_{i,j}.N/U\} \simeq^S_{Ct} (\nu m)(M\{\{\overline{m}\}_i.0/U\}|!\{m\}_j.N)$ for any S.

Proof : Similar to the proof of $P\{\tau.R/X\} \sim_{Ct} (\nu m)(P\{\overline{m}.0/X\}|!m.R)$ in [6], by induction on the structure of M.

Corollary 1. For any indexed processes M and N with $m \notin fn(M, N)$, it holds that $M\{N/U\} \simeq^{S\cup\{n\}}_{Ct} (\nu m)(M\{\{\overline{m}\}_n.0/U\}|!\{m\}_n.N)$ for any S.

Proof : It is straightforward by $\{\tau\}_{n,n}.M \simeq^{S\cup\{n\}}_{Ct} M$ and Propositions 1 and 4.

To prove the correctness of $Tr^n[\,]$, which is stated as Proposition 5, we first give the following lemma:

Lemma 1. For any $M, N \in ITPr^c$, $M \simeq^S_{Ct} N \Rightarrow M \simeq^S_{Tr} N$.

Proposition 5. For each $M \in IPr^c$,

1. $Tr^n[M]$ is an indexed triggered process;
2. $Tr^n[M] \simeq^{S\cup\{n\}}_{Ct} M$;
3. $Tr^n[M] \simeq^{S\cup\{n\}}_{Tr} M$, if M is an indexed triggered process.

Proof : 1. It is straightforward.
2. It can be proved by induction on the structure of M and using Corollary 1.
3. By Lemma 1 and Case 2.

Proposition 6 below states the relation between $\simeq^S_{Nr}$ and $\simeq^{S\cup\{n\}}_{Tr}$:

Proposition 6. For any $M, N \in IPr^c$, $M \simeq^S_{Nr} N \Rightarrow Tr^n[M] \simeq^{S\cup\{n\}}_{Tr} Tr^n[N]$, here $n \notin Index(M, N)$.

The following Lemma 2 and Lemma 3 are necessary to the proof of Proposition 7.

Lemma 2. For any $M, N \in IPr^c$, $Tr^n[M] \simeq^{S\cup\{n\}}_{Tr} Tr^n[N] \Rightarrow M \simeq^{S\cup\{n\}}_{Ct} N$, here $n \notin Index(M, N)$.

Lemma 3. For any $M, N \in IPr^c$, $M \simeq^{S\cup\{n\}}_{Ct} N \Rightarrow M \simeq^S_{Ct} N$, here $n \notin Index(M, N)$.

Proof : It is clear since $n \notin Index(M, N)$.
Now we get the relation between $\simeq^{S\cup\{n\}}_{Tr}$ and $\simeq^S_{Ct}$ as follows:

Proposition 7. For any $M, N \in IPr^c$, $Tr^n[M] \simeq^{S\cup\{n\}}_{Tr} Tr^n[N] \Rightarrow M \simeq^S_{Ct} N$, here $n \notin Index(M, N)$.

Proof : By Lemmas 2 and 3.
The following proposition is the main result of this section, which states the equivalence between indexed context bisimulation, indexed normal bisimulation and indexed triggered bisimulation.

Proposition 8. For any $M, N \in IPr^c$, $M \simeq^S_{Nr} N \Leftrightarrow Tr^n[M] \simeq^{S\cup\{n\}}_{Tr} Tr^n[N] \Leftrightarrow M \simeq^S_{Ct} N$, here $n \notin Index(M, N)$.

Proof : By Propositions 3, 6 and 7.

For indexed triggered processes, the above proposition can be simplified as Corollary 2.

Lemma 4. For any $M, N \in ITPr^c$, $M \simeq^{S\cup\{n\}}_{Tr} N \Rightarrow M \simeq^S_{Tr} N$, here $n \notin Index(M, N)$.

Proof : It is clear since $n \notin Index(M, N)$.

Corollary 2. For any M, $N \in ITPr^c$, $M \simeq^S_{Nr} N \Leftrightarrow M \simeq^S_{Tr} N \Leftrightarrow M \simeq^S_{Ct} N$.

Proof : By Proposition 8, $M \simeq^S_{Nr} N \Leftrightarrow Tr^n[M] \simeq^{S\cup\{n\}}_{Tr} Tr^n[N] \Leftrightarrow M \simeq^S_{Ct} N$, here $n \notin Index(M,N)$. Since M, $N \in ITPr^c$, $M \simeq^{S\cup\{n\}}_{Tr} Tr^n[M] \simeq^{S\cup\{n\}}_{Tr} Tr^n[N] \simeq^{S\cup\{n\}}_{Tr} N$. By Lemma 4, we have $M \simeq^S_{Tr} N$.

Sangiorgi [6] proved that barbed equivalence coincides with context bisimulation. We generalize this result to our indexed process calculus. In the following, we first present an indexed variant of barbed equivalence called indexed reduction bisimulation and then give the equivalence between indexed reduction bisimulation, indexed context bisimulation and indexed normal bisimulation. This result shows that all our indexed bisimulations are same and capture the essential of equivalence of indexed processes.

Definition 9. Let M, N be two indexed processes, and $S \subseteq I$ be an index set, we write $M \simeq^S_{Rd} N$, if there is a symmetric relation R and $K\ R\ L$ implies:

(1) $K|M\ R\ L|M$ for any indexed process M;
(2) whenever $K \stackrel{\varepsilon,S}{\Longrightarrow} K'$, there exists L' such that $L \stackrel{\varepsilon,S}{\Longrightarrow} L'$ and $K'\ R\ L'$;
(3) whenever $K \stackrel{\{\tau\}_{i,j},S}{\Longrightarrow} K'$, here $(i,j) \notin \{(k,k)|k \in S\}$, there exists L' such that $L \stackrel{\{\tau\}_{i,j},S}{\Longrightarrow} L'$ and $K'\ R\ L'$.

We say that M and N are indexed reduction bisimilar w.r.t. S if $M \simeq^S_{Rd} N$. Since $\simeq^S_{Ct}$ is equivalent to $\simeq^S_{Nr}$, the following proposition states that $\simeq^S_{Ct}$, $\simeq^S_{Nr}$ and $\simeq^S_{Rd}$ are same.

Proposition 9. For any M, $N \in IPr^c$, $M \simeq^S_{Ct} N \Rightarrow M \simeq^S_{Rd} N \Rightarrow M \simeq^S_{Nr} N$.

In [1], the concept of indexed reduction bisimulation was used to give a uniform equivalence for different process calculi.

4 The Equivalence Between Bisimulations in Higher Order π-Calculus

4.1 Strong Context Bisimulation Coincides with Strong Normal Bisimulation

The equivalence between strong context bisimulation and strong normal bisimulation can be derived by the mapping to indexed triggered process and the equivalence between indexed bisimulations.

For example, let us see the following two processes:

$P = (\nu a)(\overline{a}\langle\overline{b}.0\rangle.0|a(X).X)$;
$Q = (\nu a)(\overline{a}\langle 0\rangle.0|a(X).\overline{b}.0)$.

They are clearly strong context bisimilar. However, their triggered mappings are not strong triggered bisimilar. Indeed, the mapping of Q is $(\nu a)((\nu m)$

$(\overline{a}\langle\overline{m}.0\rangle.0|!m.0)|a(X).\overline{b}.0)$, after the communication between $\overline{a}$ and a, the residual process can perform action $\overline{b}$ without using silent tau actions, whereas the mapping of P is $(\nu a)((\nu m)(\overline{a}\langle\overline{m}.0\rangle.0|!m.\overline{b}.0)|a(X).X)$, and to match this behavior, one has to go through a trigger and this therefore requires some form of weak transition. Hence the proof strategy in [6, 7] cannot be generalized to the case of strong bisimulation.

In our approach, we first consider the indexed version of P and Q:

$$\{P\}_0 = (\nu a)(\{\overline{a}\langle\{\overline{b}\}_0.0\rangle\}_0.0|\{a(X)\}_0.X);$$
$$\{Q\}_0 = (\nu a)(\{\overline{a}\langle 0\rangle\}_0.0|\{a(X)\}_0.\{\overline{b}\}_0.0).$$

It is clearly $\{P\}_0 \simeq^{\emptyset}_{Ct} \{Q\}_0$. Now the indexed triggered mapping of $\{P\}_0$ is $Tr^n[\{P\}_0] = (\nu a)((\nu m)(\{\overline{a}\langle\{\overline{m}\}_n.0\rangle\}_0.0|!\{m\}_n.\{\overline{b}\}_0.0)|\{a(X)\}_0.X)$, and the indexed triggered mapping of $\{Q\}_0$ is $Tr^n[\{Q\}_0] = (\nu a)((\nu m)(\{\overline{a}\langle\{\overline{m}\}_n.0\rangle\}_0.0|$ $!\{m\}_n.0)|\{a(X)\}_0.\{\overline{b}\}_0.0)$. Unlike the un-indexed case, $Tr^n[\{P\}_0]$ and $Tr^n[\{Q\}_0]$ are indexed triggered bisimilar w.r.t. $S = \{n\}$. For example, let us consider the transition: $Tr^n[\{P\}_0] \stackrel{\{\tau\}_{0,0}}{\longrightarrow} (\nu a)((\nu m)(0|!\{m\}_n.\{\overline{b}\}_0.0|\{\overline{m}\}_n.0)) \stackrel{\{\tau\}_{n,n}}{\longrightarrow} (\nu a)((\nu m)$ $(0|\{\overline{b}\}_0.0|!\{m\}_n.\{\overline{b}\}_0.0|0)) \stackrel{\{\overline{b}\}_0}{\longrightarrow} (\nu a)((\nu m)(0|0|!\{m\}_n.\{\overline{b}\}_0.0|0))$. Since we neglect indexed tau action of the form $\{\tau\}_{n,n}$ in the definition of $\simeq^{\{n\}}_{Tr}$, we have a matching transition $Tr^n[\{Q\}_0] \stackrel{\{\tau\}_{0,0}}{\longrightarrow} (\nu a)((\nu m)(0|!\{m\}_n.0|\{\overline{b}\}_0.0)) \stackrel{\{\overline{b}\}_0}{\longrightarrow} (\nu a)((\nu m)$ $(0|!\{m\}_n.0|0))$. Hence $Tr^n[\{P\}_0]$ and $Tr^n[\{Q\}_0]$ are bisimilar. Formally, we have $Tr^n[\{P\}_0] \simeq^{\{n\}}_{Tr} Tr^n[\{Q\}_0]$. Similarly, we can further build the relation between $\simeq^{\{n\}}_{Tr}$ and $\simeq^{\emptyset}_{Nr}$: $Tr^n[\{P\}_0] \simeq^{\{n\}}_{Tr} Tr^n[\{Q\}_0] \Leftrightarrow \{P\}_0 \simeq^{\emptyset}_{Nr} \{Q\}_0$.

In this section, we will show that $P \sim_{Nr} Q \Rightarrow \{P\}_0 \simeq^{\emptyset}_{Nr} \{Q\}_0$ and $\{P\}_0 \simeq^{\emptyset}_{Ct}$ $\{Q\}_0 \Rightarrow P \sim_{Ct} Q \Rightarrow P \sim_{Nr} Q$. Since $\{P\}_0 \simeq^{\emptyset}_{Ct} \{Q\}_0 \Leftrightarrow \{P\}_0 \simeq^{\emptyset}_{Nr} \{Q\}_0$ by Proposition 8, the equivalence between $P \sim_{Nr} Q$ and $P \sim_{Ct} Q$ is obvious.

Now we prove that strong context bisimulation and strong normal bisimulation coincide, which was presented in [7] as an open problem.

Firstly, we introduce the concept of strong indexed context equivalence, strong indexed normal equivalence and strong indexed triggered equivalence.

Definition 10. Strong indexed context equivalence.
Let $P, Q \in Pr^c$, we write $P \sim^i_{Ct} Q$, if $\{P\}_k \simeq^{\emptyset}_{Ct} \{Q\}_k$ for some index k. As we defined before, here $\{P\}_k$ denotes indexed process with the same given index k on every prefix in P.

Definition 11. Strong indexed normal equivalence.
Let $P, Q \in Pr^c$, we write $P \sim^i_{Nr} Q$, if $\{P\}_k \simeq^{\emptyset}_{Nr} \{Q\}_k$ for some index k.

Definition 12. Strong indexed triggered equivalence.
Let $P, Q \in Pr^c$, we write $P \sim^{i,\{n\}}_{Tr} Q$, if $Tr^n[\{P\}_k] \simeq^{\{n\}}_{Tr} Tr^n[\{Q\}_k]$ for some index k with $k \neq n$.

The following lemma states that strong normal bisimulation implies strong indexed normal equivalence.

Lemma 5. For any $P, Q \in Pr^c$, $P \sim_{Nr} Q \Rightarrow P \sim^i_{Nr} Q$.

Now, the equivalence between $\sim_{Nr}$ and $\sim_{Ct}$ can be given.

Proposition 10. For any $P, Q \in Pr^c$ and any index n, $P \sim_{Nr} Q \Leftrightarrow P \sim^i_{Nr} Q \Leftrightarrow P \sim^{i,\{n\}}_{Tr} Q \Leftrightarrow P \sim^i_{Ct} Q \Leftrightarrow P \sim_{Ct} Q$.

Proof : Firstly, it is easy to prove $P \sim^i_{Ct} Q \Rightarrow P \sim_{Ct} Q \Rightarrow P \sim_{Nr} Q$. By Lemma 5, $P \sim_{Nr} Q \Rightarrow P \sim^i_{Nr} Q$. Hence $P \sim^i_{Ct} Q \Rightarrow P \sim_{Ct} Q \Rightarrow P \sim_{Nr} Q \Rightarrow P \sim^i_{Nr} Q$. By Proposition 8, we have $P \sim^i_{Nr} Q \Leftrightarrow P \sim^{i,\{n\}}_{Tr} Q \Leftrightarrow P \sim^i_{Ct} Q$ for any index n. Therefore the proposition holds.

Moreover, we can define strong indexed reduction equivalence $\sim^i_{Rd}$ as follows: let $P, Q \in Pr^c$, we write $P \sim^i_{Rd} Q$, if $\{P\}_k \simeq^{\emptyset}_{Rd} \{Q\}_k$ for some index k. By Propositions 9 and 10, we know that $\sim^i_{Rd}$ coincides with $\sim_{Nr}$ and $\sim_{Ct}$.

4.2 Weak Context Bisimulation Coincides with Weak Normal Bisimulation

Based on the equivalence between indexed bisimulations, we can give an alternative proof for the equivalence between weak context bisimulation and weak normal bisimulation.

Definition 13. Weak indexed context equivalence.
Let $P, Q \in Pr^c$, we write $P \approx^i_{Ct} Q$, if $\{P\}_k \simeq^I_{Ct} \{Q\}_k$ for some index k, here I is the full index set.

Definition 14. Weak indexed normal equivalence.
Let $P, Q \in Pr^c$, we write $P \approx^i_{Nr} Q$, if $\{P\}_k \simeq^I_{Nr} \{Q\}_k$ for some index k, here I is the full index set.

Definition 15. Weak indexed triggered equivalence.
Let $P, Q \in Pr^c$, we write $P \approx^i_{Tr} Q$, if $Tr^n[\{P\}_k] \simeq^I_{Tr} Tr^n[\{Q\}_k]$ for some indices k and n, here $k \neq n$ and I is the full index set.

Lemma 6. For any $P, Q \in Pr^c$, $P \approx_{Nr} Q \Rightarrow P \approx^i_{Nr} Q$.

Proposition 11. For any $P, Q \in Pr^c$, $P \approx_{Nr} Q \Leftrightarrow P \approx^i_{Nr} Q \Leftrightarrow P \approx^i_{Tr} Q \Leftrightarrow P \approx^i_{Ct} Q \Leftrightarrow P \approx_{Ct} Q$.

Proof : By Proposition 8, it is easy to get $P \approx^i_{Nr} Q \Rightarrow P \approx^i_{Tr} Q \Rightarrow P \approx^i_{Ct} Q \Rightarrow P \approx_{Ct} Q \Rightarrow P \approx_{Nr} Q$. By Lemma 6, $P \approx_{Nr} Q \Rightarrow P \approx^i_{Nr} Q$, therefore the proposition holds.

Similarly, we can define weak indexed reduction equivalence $\approx^i_{Rd}$ as follows: let $P, Q \in Pr^c$, we write $P \approx^i_{Rd} Q$, if $\{P\}_k \simeq^I_{Rd} \{Q\}_k$ for some index k. By Propositions 9 and 11, $\approx^i_{Rd}$ coincides with $\approx_{Nr}$ and $\approx_{Ct}$.

In [6, 7], the proposition: $P \approx_{Nr} Q \Leftrightarrow Tr[P] \approx_{Tr} Tr[Q] \Leftrightarrow P \approx_{Ct} Q$ was proved, where $Tr[\ ]$ is the triggered mapping and $\approx_{Tr}$ is the weak triggered bisimulation. In fact, this proposition can be get from Proposition 11. Firstly by

Proposition 11, we have $P \approx_{Nr} Q \Leftrightarrow Tr^n[\{P\}_k] \simeq^I_{Tr} Tr^n[\{Q\}_k] \Leftrightarrow P \approx_{Ct} Q$. Secondly, we can prove that $Tr[P] \approx_{Tr} Tr[Q] \Leftrightarrow Tr^n[\{P\}_k] \simeq^I_{Tr} Tr^n[\{Q\}_k]$. Hence $P \approx_{Nr} Q \Leftrightarrow Tr[P] \approx_{Tr} Tr[Q] \Leftrightarrow P \approx_{Ct} Q$ is a corollary of Proposition 11. But for the strong case, the claim: $P \sim_{Nr} Q \Leftrightarrow Tr[P] \sim_{Tr} Tr[Q] \Leftrightarrow P \sim_{Ct} Q$ does not hold. For example, let $P = (\nu a)(\overline{a}\langle \overline{b}.0\rangle.0|a(X).X)$ and $Q = (\nu a)(\overline{a}\langle 0\rangle.0|a(X).\overline{b}.0)$, then $P \sim_{Nr} Q$, $P \sim_{Ct} Q$ and $Tr[P] \not\sim_{Tr} Tr[Q]$. Hence $P \sim_{Nr} Q \not\Leftrightarrow Tr[P] \sim_{Tr} Tr[Q] \not\Leftrightarrow P \sim_{Ct} Q$. This also shows that we cannot prove the equivalence between strong context bisimulation and strong normal bisimulation by the original technique of triggered mapping.

5 Conclusions

To prove the equivalence between context bisimulation and normal bisimulation, this paper proposed an indexed higher order π-calculus. In fact, this indexed calculus can also be viewed as a model of distributed computing, where indices represent locations, indexed action $\{\alpha\}_i$ represents an input/output action α performed in location i, and $\{\tau\}_{i,j}$ represents a communication between locations i and j. There are a few results on bisimulations for higher order π-calculus. In [6], context bisimulation and normal bisimulation were compared with barbed equivalence. In [3], authors proved a correspondence between weak normal bisimulation and a variant of barbed equivalence, called contextual barbed equivalence. In [4] an alternative proof of the correspondence between context bisimulation and barbed equivalence was given. It would be interesting to understand whether our concept of indexed processes and indexed bisimulations can be helpful to study the relation between bisimulations in the framework of other higher order concurrency languages.

References

1. Z. Cao. A uniform reduction equivalence for process calculi, In Proc. APLAS'04, Lecture Notes in Computer Science 3302, 179-195. Springer-Verlag, 2004.
2. A. Jeffrey, J. Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. Theoretical Computer Science. 323:1-48, 2004.
3. A. Jeffrey, J. Rathke. Contextual equivalence for higher-order π-calculus revisited. Logical Methods in Computer Science, 1(1:4):1-22, 2005.
4. Y. Li, X. Liu: Towards a theory of bisimulation for the higher-order process calculi. Journal of Computer Science and Technology. 19(3): 352-363, 2004.
5. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Part I and II). Information and Computation, 100:1-77, 1992.
6. D. Sangiorgi. Expressing mobility in process algebras: first-order and higher-order paradigms, Ph.D thesis, University of Einburgh, 1992.
7. D. Sangiorgi. Bisimulation in higher-order calculi, Information and Computation, 131(2):141-178, 1996.
8. D. Sangiorgi, D. Walker. The π-calculus: a theory of mobile processes, Cambridge University Press, 2001.
9. B. Thomsen. Plain CHOCS, a second generation calculus for higher order processes, Acta Information, 30:1-59, 1993.

Register Allocation After Classical SSA Elimination is NP-Complete

Fernando Magno Quintão Pereira and Jens Palsberg

UCLA, University of California, Los Angeles

Abstract. Chaitin proved that register allocation is equivalent to graph coloring and hence NP-complete. Recently, Bouchez, Brisk, and Hack have proved independently that the interference graph of a program in static single assignment (SSA) form is chordal and therefore colorable in linear time. Can we use the result of Bouchez et al. to do register allocation in polynomial time by first transforming the program to SSA form, then performing register allocation, and finally doing the classical SSA elimination that replaces ϕ-functions with copy instructions? In this paper we show that the answer is no, unless P = NP: register allocation after classical SSA elimination is NP-complete. Chaitin's proof technique does not work for programs after classical SSA elimination; instead we use a reduction from the graph coloring problem for circular arc graphs.

1 Introduction

In Section 1.1 we define three central notions that we will use in the paper: the core register allocation problem, static single assignment (SSA) form, and post-SSA programs. In Section 1.2 we explain why recent results on programs in SSA form might lead one to speculate that we can solve the core register allocation problem in polynomial time. Finally, in Section 1.3 we outline our result that register allocation is NP-complete for post-SSA programs produced by the classical approach that replaces ϕ-functions with copy instructions.

1.1 Background

Register Allocation. In a compiler, register allocation is the problem of mapping temporaries to machine registers. In this paper we will focus on:

> **Core register allocation problem:**
> Instance: a program P and a number K of available registers.
> Problem: can each of the temporaries of P be mapped to one of the K registers such that temporary variables with interfering live ranges are assigned to different registers?

Notice that K is part of the input to the problem. Fixing K would correspond to the register allocation problem solved by a compiler for a fixed architecture. Our core register allocation problem is related to the kind of register allocation problem solved by gcc; the problem does not make assumptions about the number of registers in the target architecture.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 79–93, 2006.
© Springer-Verlag Berlin Heidelberg 2006

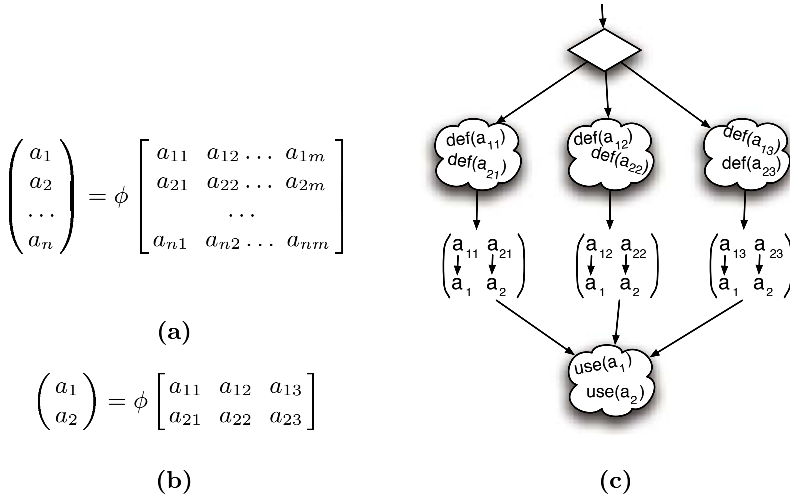


Fig. 1. (a) ϕ-functions represented as a matrix equation. (b) Matrix equation representing two ϕ-functions and three possible execution paths. (c) Control flow graph illustrating the semantics of ϕ-functions.

Chaitin et al. [8] showed that the core register allocation problem is NP-complete by a reduction from the graph coloring problem. The essence of Chaitin et al.'s proof is that every graph is the interference graph of some program.

SSA form. Static single assignment (SSA) form [21] is an intermediate representation used in many compilers, including gcc version 4. If a program is in SSA form, then every variable is assigned exactly once, and each use refers to exactly one definition. A compiler such as gcc version 4 translates a given program to SSA form and later to an executable form.

SSA form uses ϕ-functions to join the live ranges of different names that represent the same value. We will describe the syntax and semantics of ϕ-functions using the matrix notation introduced by Hack et al. [17]. Figure 1 (a) outlines the general representation of a ϕ-matrix. And Figure 1 (c) gives the intuitive semantics of the matrix shown in Figure 1 (b).

An equation such as $V = \phi M$, where V is a n-dimensional vector, and M is a $n \times m$ matrix, contains n ϕ-functions such as $a_i \leftarrow \phi(a_{i1}, a_{i2}, \dots a_{im})$. Each possible execution path has a corresponding column in the ϕ-matrix, and adds one parameter to each ϕ-function. The ϕ symbol works as a multiplexer. It will assign to each element a_i of V an element a_{ij} of M, where j is determined by the actual path taken during the program's execution.

All the ϕ-functions are evaluated simultaneously at the beginning of the basic block where they are located. As noted by Hack et al. [17], the live ranges of temporaries in the same column of a ϕ-matrix overlap, while the live ranges of temporaries in the same row do not overlap. Therefore, we can allocate the same register to two temporaries in the same row. For example, Figure 2 shows a program, its SSA version, and the program after classical SSA elimination.

If the control flow reaches block 2 from block 1, $\phi(v_{11}, v_{12})$ will return v_{11}; v_{12} being returned otherwise. Variables i_2 and v_{11} do not interfere. In contrast, the variables v_{11} and i_1 interfere because both are alive at the end of block 1.

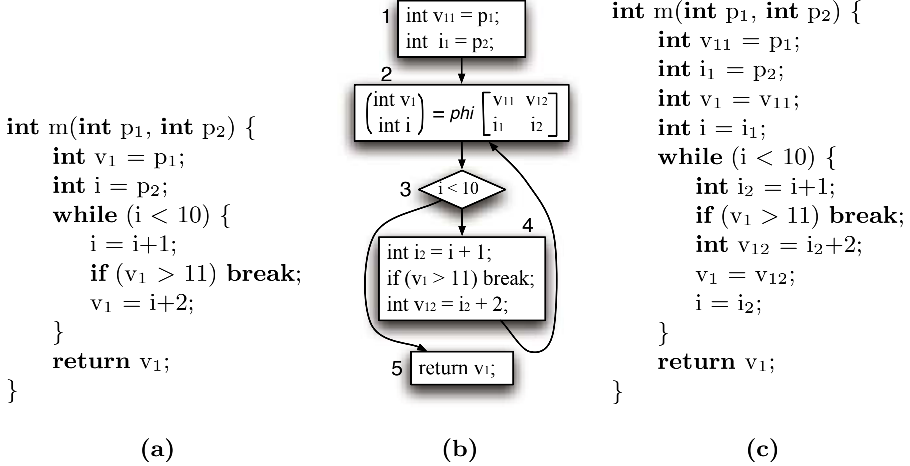


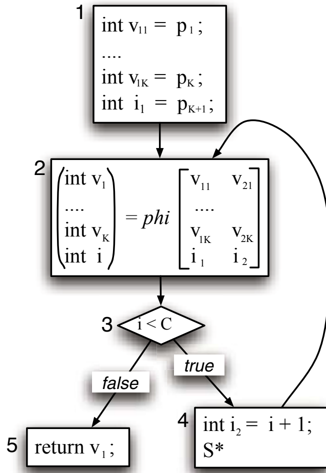
Fig. 2. (a) A program. (b) The same program in SSA form. (c) The program after classical SSA elimination.

Post-SSA programs. SSA form simplifies many analyses that are performed on the control flow graph of programs. However, traditional instruction sets do not implement ϕ-functions [10]. Thus, in order to generate executable code, compilers have a phase called *SSA-elimination* in which ϕ-functions are destroyed. Henceforth, we will refer to programs after SSA elimination as *post-SSA* programs.

The classical approach to SSA-elimination replaces the ϕ-functions with copy instructions [1, 5, 7, 10, 18, 20]. For example, consider $v_1 = \phi(v_{11}, ..., v_{1m})$ in block b. The algorithm explained by Appel [1] adds at the end of each block i that precedes b, one copy instruction such as $v_1 = v_{1i}$.

In this paper we concentrate on SSA programs whose control flow graphs have the structure outlined in Figure 3 (a). The equivalent post-SSA programs, generated by the classical approach to SSA-elimination, are given by the grammar in Figure 3 (b). We will say that a program generated by the grammar in Figure 3 (b) is a *simple* post-SSA program. For example, the program in Figure 2(c) is a simple post-SSA program. A simple post-SSA program contains a single loop. Just before the loop, and at the end of it (see Figure 3 (b)), the program contains copy instructions that correspond to the elimination of a ϕ-matrix such as:

$$\begin{pmatrix} v_1 \\ \dots \\ v_K \\ i \end{pmatrix} = \phi \begin{bmatrix} v_{11} & v_{21} \\ & \dots \\ v_{1K} & v_{2K} \\ i_1 & i_2 \end{bmatrix}$$



(a)

```
P ::= int m(int p1, ..., int pK+1) {
          int v11 = p1; ...; int v1K = pK;
          int i1 = pK+1;
          int v1 = v11; ...; int vK = v1K;
          int i = i1;
          while (i< C) {
              int i2 = i+1;
              S*
              v1 = v21; ...; vK = v2K;
              i = i2;
          }
          return v1;
      }
S ::= int vj = i+C;
    | vj = vk;
    | if (vj > C) break;
C  ranges over integer constants
```

(b)

Fig. 3. (a) Control flow representation of simple SSA programs. (b) The grammar for simple post-SSA programs.

1.2 Programs in SSA-Form Have Chordal Interference Graphs

The core register allocation problem is NP-complete and a compiler can transform a given program into SSA form in cubic time [9]. Thus we might expect that the core register allocation problem for programs in SSA form is NP-complete. However, that intuition would be wrong, unless P = NP, as demonstrated by the following result.

In 2005, Brisk et al. [6] proved that *strict* programs in SSA form have *perfect* interference graphs; independently, Bouchez [4] and Hack [16] proved the stronger result that strict programs in SSA form have *chordal* interference graphs. In a strict program, every path from the initial block to the use of a variable v passes through a definition of v [7]. The proofs presented in [4, 16] rely on two well-known facts: (i) the chordal graphs are the intersection graphs of subtrees in trees [14], and (ii) live ranges in an SSA program are subtrees of the dominance tree [16].

We can color a chordal graph in linear time [15] so we can solve the core register allocation problem for programs in SSA form in linear time. Thus, the transformation to SSA form seemingly maps an NP-complete problem to a polynomial-time problem in polynomial time! The key to understanding how such a transformation is possible lies in the following observation. Given a program P, its SSA-form version P', and a number of registers K, the core register allocation problem (P, K) is not equivalent to (P', K). While we can map a (P, K)-solution to a (P', K)-solution, we can not necessarily map a (P', K)-solution to a (P, K)-solution. The SSA transformation splits the live ranges of temporaries in P in such a way that P' may need fewer registers than P.

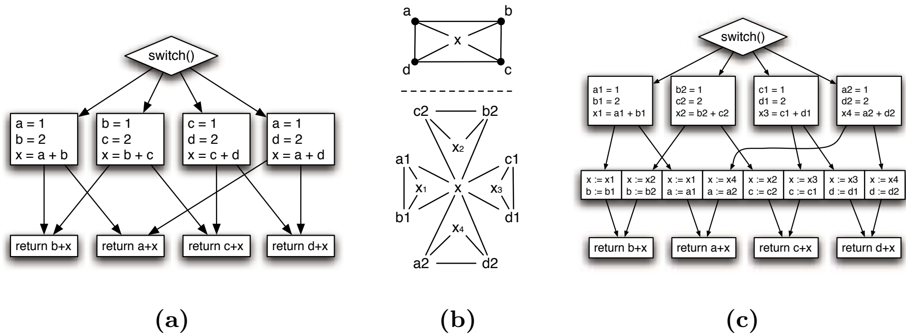
Given that the core register allocation problem for programs in SSA form can be solved in polynomial time and given that a compiler can do classical SSA elimination in linear time, we might expect that the core register allocation problem after classical SSA elimination is in polynomial time. In this paper we show that also that intuition would be wrong!

1.3 Our Result

We prove that the core register allocation problem for simple post-SSA programs is NP-complete. Our result has until now been a commonly-believed folk theorem without a published proof. The recent results on register allocation for programs in SSA form have increased the interest in a proof. Our result implies that the core register allocation problem for post-SSA programs is NP-complete for any language with loops or with jumps that can implement loops.

The proof technique used by Chaitin et al. [8] does not work for post-SSA programs. Chaitin et al.'s proof constructs programs from graphs, and if we transform those programs to SSA form and then post-SSA form, we can color the interference graph of each of the post-SSA programs with just three colors. For example, in order to represent C_4, their technique would generate the graph in the upper part of Figure 4 (b). The minimal coloring of such graph can be trivially mapped to a minimal coloring of C_4, by simply deleting node x. Figure 4 (a) shows the control flow graph of the program generated by Chaitin et al.'s proof technique, and Figure 4 (c) shows the program in post-SSA form. The interference graph of the transformed program is shown in the lower part of Figure 4 (b); that graph is chordal, as expected.

We prove our result using a reduction from the graph coloring problem for circular arc graphs, henceforth called simply *circular graphs*. A circular graph can be depicted as a set of intervals around a circle (e.g. Figure 5 (a)). The idea of our proof is that the live ranges of the variables in the loop in a simple post-SSA program form a circular graph. From a circular graph and an integer K we build a simple post-SSA program such that the graph is K-colorable if



(a) **(b)** **(c)**

Fig. 4. (a) Chaitin et al.'s program to represent C_4. (b) The interference graph of the original program (top) and of the program in SSA form (bottom). (c) The program of Chaitin et al. in SSA form.

and only if we can solve the core register allocation problem for the program and $K+1$ registers. Our reduction proceeds in two steps. In Section 2 we define the notion of SSA-circular graphs and we show that the coloring problem for SSA-circular graphs is NP-complete. In Section 3 we present a reduction from coloring of SSA-circular graphs to register allocation for simple post-SSA programs. An SSA-circular graph is a special case of a circular graph in which some of the intervals come in pairs that correspond to the copy instructions at the end of the loop in a simple post-SSA program. From a circular graph we build an SSA-circular graph by splitting some arcs. By adding new intervals at the end of the loop, we artificially increase the color pressure at that point, and ensure that two intervals that share an extreme point receive the same color. In Section 4 we give a brief survey of related work on complexity results for a variety of register allocation problems, and in Section 5 we conclude.

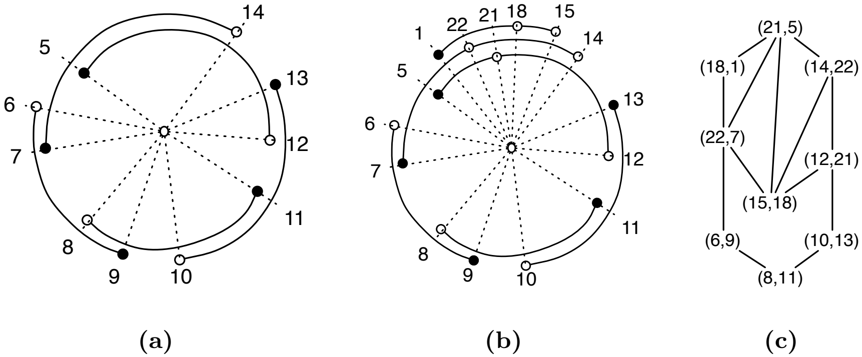


Fig. 5. (a) C_5 represented as a set of intervals. (b) The set of intervals that represent $W = \mathcal{F}(C_5, 3)$. (c) W represented as a graph.

Recently, Hack et al. [17] presented an SSA-elimination algorithm that does not use move instructions to replace ϕ-functions. Instead, Hack et al.'s algorithm uses xor instructions to permute the values of the parameters of the ϕ-functions in a way that preserves both the semantics of the original program and the chordal structure of the interference graph, without demanding extra registers. As a result, register allocation after the Hack et al.'s approach to SSA elimination is in polynomial time. In contrast, register allocation after the classical approach to SSA elimination is NP-complete.

2 From Circular Graphs to SSA-Circular Graphs

Let N denote the set of positive, natural numbers $\{1, 2, 3, \ldots\}$. A *circular graph* is an undirected graph given by a finite set of vertices $V \subseteq N \times N$, such that $\forall d \in N : (d, d) \notin V$ and $\forall (d, u), (d', u') \in V : d = d' \Leftrightarrow u = u'$. We sometimes

refer to a vertex (d,u) as an *interval*, and we call d,u *extreme points*. The set of vertices of a circular graph defines the edges implicitly, as follows. Define

$$b : N \times N \rightarrow \text{ finite unions of intervals of the real numbers}$$
$$b(d,u) = \begin{cases}]d,u[& \text{if } d < u \\ ]0,u[\cup]d,\infty[& \text{if } d > u. \end{cases}$$

Two vertices $(d,u),(d',u')$ are connected by an edge if and only if $b(d,u) \cap b(d',u') \neq \emptyset$. We use $\mathcal{V}$ to denote the set of such representations of circular graphs. We use $\max(V)$ to denote the largest number used in V, and we use $\min(V)$ to denote the smallest number used in V. We distinguish three subsets of vertices of a circular graph, V_l, V_i and V_z:

$$V_i = \{ (d,u) \in V \mid d < u \}$$
$$V_l = \{ (d,u) \in V \mid d > u \}$$
$$V_z = \{ (d,y) \in V_i \mid \exists (y,u) \in V_l \}$$

Notice that $V = V_i \cup V_l$, $V_i \cap V_l = \emptyset$, and $V_z \subseteq V_i$.

Figure 5 (a) shows a representation of $C_5 = (\{a,b,c,d,e\},\{ab,bc,cd,de,ea\})$ as a collection of intervals, where $a = (14,7)$, $b = (6,9)$, $c = (8,11)$, $d = (10,13)$, $e = (12,5)$, $V_i = \{b,c,d\}$, $V_l = \{a,e\}$, and $V_z = \emptyset$. Intuitively, when the intervals of the circular graph are arranged around a circle, overlapping intervals determine edges between the corresponding vertices.

An SSA-circular graph W is a circular graph with two additional properties:

$$\forall (y,u) \in W_l : \exists d \in N : (d,y) \in W_z \quad (1)$$
$$\forall (d,u) \in W_i \setminus W_z : \forall (d',u') \in W_l : u < d' \quad (2)$$

We use $\mathcal{W}$ to denote the set of SSA-circular graphs.

Let W be an SSA-circular graph. Property (1) says that for each interval in W_l there is an interval in W_z so that these intervals share an extreme point y. In Section 3 it will be shown that the y points represent copy instructions used to propagate the parameters of ϕ-functions. Henceforth, the y points will be called *copy points*. Figure 5 (b) shows $W \in \mathcal{W}$ as an example of SSA-circular graph. $W_l = \{(18,1),(21,5),(22,7)\}$, $W_i = \{(6,9),(8,11),(10,13)\} \cup W_z$, and $W_z = \{(15,18),(12,21),(14,22)\}$. Notice that for every interval $(y,u) \in W_l$, there is an interval $(d,y) \in W_z$. Figure 5 (c) exhibits W using the traditional representation of graphs.

Let $n = |V_l|$. We will now define a mapping $\mathcal{F}$ on pairs (V,K):

$$\mathcal{F} : \mathcal{V} \times N \rightarrow \mathcal{W}$$
$$\mathcal{F}(V,K) = V_i \cup \mathcal{G}(V_l, K, \max(V))$$
$$\mathcal{G} : \mathcal{V} \times N \times N \rightarrow \mathcal{V}$$

$$\mathcal{G}(\{ (d_i,u_i) \mid i \in 1..n \}, K, m) = \{ (m+i, m+K+i) \mid i \in 1..K-n \} \quad (3)$$
$$\cup \{ (m+K+i, i) \mid i \in 1..K-n \} \quad (4)$$
$$\cup \{ (d_i, m+2K+i) \mid i \in 1..n \} \quad (5)$$
$$\cup \{ (m+2K+i, u_i) \mid i \in 1..n \} \quad (6)$$

Given V, the function $\mathcal{F}$ splits each interval of V_l into two nonadjacent intervals that share an extreme point: $(d, y) \in W_z$, and $(y, u) \in W_l$. We call those intervals the *main* vertices. Given V, the function $\mathcal{F}$ also creates $2(K - n)$ new intervals, namely $K - n$ pairs of intervals such that the two intervals of each pair are nonadjacent and share an extreme point: $(d, y) \in W_z$, and $(y, u) \in W_l$. We call those intervals the *auxiliary* vertices. Figures 5 (b) and 5 (c) represent $\mathcal{F}(C_5, 3)$, and Figure 6 outlines the critical points between $m = \max(V)$ and $K - n$.

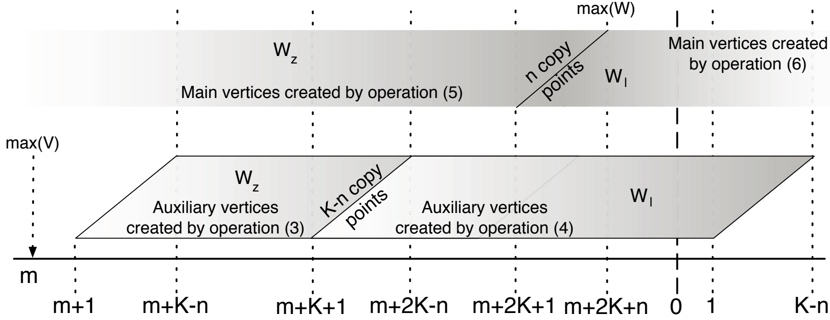


Fig. 6. The critical points created by $\mathcal{F}(V, K)$

Lemma 1. *If V is a circular graph and $\min(V) > K$, then $\mathcal{F}(V, K)$ is an SSA-circular graph.*

Proof. Let $W = \mathcal{F}(V, K)$. Notice first that W is a circular graph because the condition $\min(V) > K$ ensures that rules (4) and (6) define vertices that don't share any extreme points. To see that W is an SSA-circular graph let us consider in turn the two conditions (1) and (2). Regarding condition (1), W_l consists of the sets defined by (4), (6), while W_z consists of the sets defined by (3), (5), and for each $(y, u) \in W_l$, we can find $(d, y) \in W_z$. Regarding condition (2), we have that if $(d, u) \in W_i \setminus W_z$ and $(d', u') \in W_l$, then $u \leq \max(V) < \max(V) + K + 1 \leq d'$. □

Lemma 2. *If $W = \mathcal{F}(V, K)$ is K-colorable, then two intervals in $W_z \cup W_l$ that share an extreme point must be assigned the same color by any K-coloring of W.*

Proof. Let c be a K-coloring of W. Let $v_1 = (d, y)$ and $v_2 = (y, u)$ be a pair of intervals in $W_z \cup W_l$. From the definition of $\mathcal{F}(V, K)$ we have that those two intervals are not connected by an edge. The common copy point y is crossed by exactly $K - 1$ intervals (see Figure 6). Each of those $K - 1$ intervals must be assigned a different color by c so there remains just one color that c can assign to v_1 and v_2. Therefore, c has assigned the same color to v_1 and v_2. □

Lemma 3. *Suppose V is a circular graph and $\min(V) > K$. We have V is K-colorable if and only if $\mathcal{F}(V, K)$ is K-colorable.*

Proof. Let $V_l = \{ \ (d_i, u_i) \ \mid \ i \in 1..n \ \}$ and let $m = \max(V)$.

First, suppose c is a K-coloring of V. The vertices of V_l form a clique so c must use $|V_l|$ colors to color V_l. Let $n = |V_l|$. Let $\{x_1, \ldots, x_{K-n}\}$ be the set of colors *not* used by c to color V_l. We now define a K-coloring c' of $\mathcal{F}(V, K)$:

$$c'(v) = \begin{cases} c(v) & \text{if } v \in V_i \\ x_i & \text{if } v = (m+i, m+K+i), i \in 1..K-n \\ x_i & \text{if } v = (m+K+i, i), i \in 1..K-n \\ c(d_i, u_i) & \text{if } v = (d_i, m+2K+i), i \in 1..n \\ c(d_i, u_i) & \text{if } v = (m+2K+i, u_i), i \in 1..n \end{cases}$$

To see that c' is indeed a K-coloring of $\mathcal{F}(V, K)$, first notice that the colors of the main vertices don't overlap with the colors of the auxiliary vertices. Second, notice that since $\min(V) > K$, no auxiliary edge is connected to a vertex in V_i. Third, notice that since c is a K-coloring of V, the main vertices have colors that don't conflict with their neighbors in V_i.

Conversely, suppose c' is a K-coloring of $\mathcal{F}(V, K)$. We now define a K-coloring c of V:

$$c(v) = \begin{cases} c'(v) & \text{if } v \in V_i \\ c'(d_i, m+2K+i) & \text{if } v = (d_i, u_i), i \in 1..n \end{cases}$$

To see that c is indeed a K-coloring of V, notice that from Lemma 2 we have that c' assigns the same color to the intervals $(d_i, m+2K+i), (m+2K+i, u_i)$ for each $i \in 1..n$. So, since c' is a K-coloring of $\mathcal{F}(V, K)$, the vertices in V_l have colors that don't conflict with their neighbors in V_i. □

Lemma 4. *Graph coloring for SSA-circular graphs is NP-complete.*

Proof. First notice that graph coloring for SSA-circular graphs is in NP because we can verify any color assignment in polynomial time. To prove NP-hardness, we will do a reduction from graph coloring for circular graphs, which is known to be NP-complete [12, 19]. Given a graph coloring problem instance (V, K) where V is a circular graph, we first transform V into an isomorphic graph V' by adding K to all the integers used in V. Notice that $\min(V') > K$. Next we produce the graph coloring problem instance $(\mathcal{F}(V', K), K)$, and, by Lemma 3, V' is K-colorable if and only if $\mathcal{F}(V', K)$ is K-colorable. □

3 From SSA-Circular Graphs to Post-SSA Programs

We now present a reduction from coloring of SSA-circular graphs to the core register allocation problem for simple post-SSA programs. In this section we use a representation of circular graphs which is different from the one used in Section 2. We represent a circular graph by a finite list of elements of the set $\mathcal{I} = \{ \ \text{def}(j), \text{use}(j), \text{copy}(j,j'), \ \mid j, j' \in N \ \}$. Each j represents a temporary name in a program. We use ℓ to range over finite lists over $\mathcal{I}$. If ℓ is a finite list over $\mathcal{I}$ and the d-th element of ℓ is either def(j) or copy(j,j'), then we say that

j is *defined* at index d of ℓ. Similarly, if the u'th element of ℓ is either use(j') or copy(j,j'), then we say that j' is *used* at index u of ℓ. We define $\mathcal{X}$ as follows:

$$\mathcal{X} = \{ \ \ell \mid \text{for every } j \text{ mentioned in } \ell, j \text{ is defined exactly once and used exactly once} \land \text{for every copy}(j,j') \text{ in } \ell, \text{ we have } j \neq j' \ \}$$

We will use X to range over $\mathcal{X}$. The sets $\mathcal{X}$ and $\mathcal{V}$ are isomorphic; the function α is an isomorphism which maps $\mathcal{X}$ to $\mathcal{V}$:

$$\alpha \ : \ \mathcal{X} \rightarrow \mathcal{V}$$
$$\alpha(X) = \{ \ (d,u) \mid \exists j \in N : j \text{ is defined at index } d \text{ of } X \text{ and } j \text{ is used at index } u \text{ of } X \ \}$$

We define $\mathcal{Y} = \alpha^{-1}(\mathcal{W})$, and we use Y to range over $\mathcal{Y}$. The graph W shown in Figure 5 (b) is shown again in Figure 7 (a) in the new representation:

$$Y = \langle \text{use}(t), \text{use}(e), \text{def}(b), \text{use}(a), \text{def}(c), \text{use}(b), \text{def}(d), \text{use}(c), \text{def}(e), \text{use}(d), \text{def}(a), \text{def}(t_2), \text{copy}(t,t_2), \text{copy}(e,e_2), \text{copy}(a,a_2)\rangle.$$

Figure 8 presents a mapping $\mathcal{H}$ from $\mathcal{Y}$-representations of SSA-circular graphs to simple post-SSA programs. Given an interval (d,u) represented by $def(j)$ and

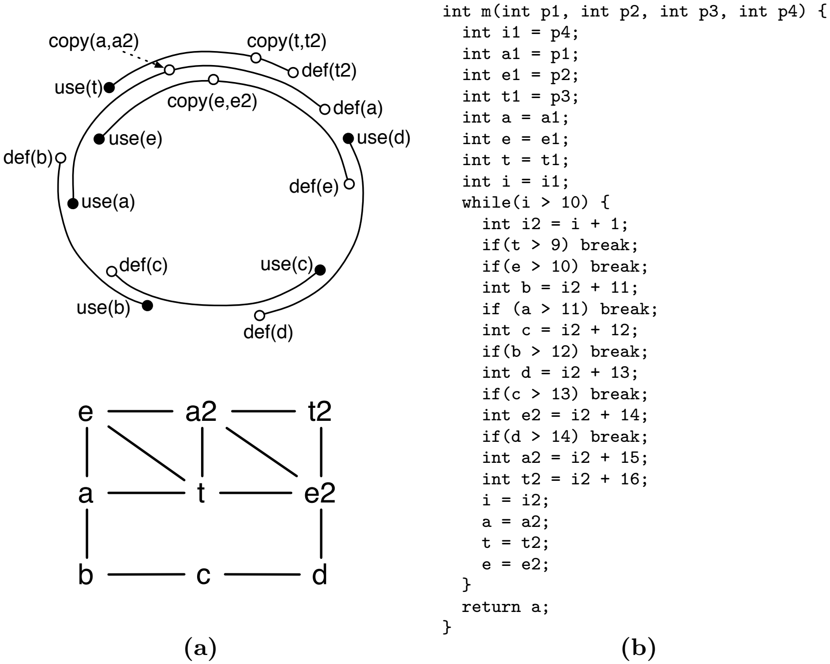


Fig. 7. (a) $Y = \mathcal{F}(C_5, 3)$ represented as a sequence of instructions and as a graph. (b) $P = \mathcal{H}(Y, 3)$.

$$
\begin{aligned}
\text{gen}(\text{def}(j)) &= \textbf{int } \mathrm{v}_j = \mathrm{i}_2 + \mathcal{C};\\
\text{gen}(\text{use}(j)) &= \textbf{if } (\mathrm{v}_j > \mathcal{C}) \textbf{ break;}\\
\text{gen}(\text{copy}(j,j')) &= \mathrm{v}_j = \mathrm{v}_{j'};\\
\mathcal{H} &: \mathcal{Y} \times N \rightarrow \text{simple post-SSA program}\\
\mathcal{H}(Y,K) &= \textbf{int } \mathrm{m}(\textbf{int } \mathrm{p}_1, \ldots, \textbf{int } \mathrm{p}_{K+1}) \; \{
\end{aligned}
$$

```
int v11 = p1; ...; int v1K = pK;
int i1 = pK+1;
int v1 = v11; ...; int vK = v1K;
int i = i1;
while (i< C) {
    int i2 = i+1;
    map(Y, gen)
    i = i2;
}
return v1;
}
```

Fig. 8. The mapping of circular graphs to simple post-SSA programs

$use(j)$, we map the initial point d to a variable definition $\mathbf{v_j} = \mathbf{i_2} + \mathcal{C}$, where $\mathbf{i_2}$ is the variable that controls the loop. We assume that all the constants are chosen to be different. The final point u is mapped to a variable use, which we implement by means of the conditional statement **if** $(\mathbf{v_j} > \mathcal{C})$ **break**. We opted for mapping uses to conditional commands because they do not change the live ranges inside the loop, and their compilation do not add extra registers to the final code. An element of the form $copy(j, j')$, which is mapped to the assignment $j = j'$, is used to simulate the copy of one of the parameters of a ϕ-function, after classical SSA elimination. Figure 7 (b) shows the program $P = \mathcal{H}(Y, 3)$, where $Y = \mathcal{F}(C_5, 3)$.

Lemma 5. *We can color an SSA-circular graph Y with K colors if and only if we can solve the core register allocation problem for $\mathcal{H}(Y,K)$ and $K+1$ registers.*

Proof. First, assume Y has a K-coloring. The intervals in Y match the live ranges in the loop of $\mathcal{H}(Y,K)$, except for the control variables i, and i_2, which have nonoverlapping live ranges. Therefore, the interference graph for the loop can be colored with $K+1$ colors. The live ranges of the variables declared outside the loop form an interval graph of width $K+1$. We can extend the $K+1$-coloring of that interval graph to a $K+1$-coloring of the entire graph in linear time.

Now, assume that there is a solution of the core register allocation problem for $\mathcal{H}(Y,K)$ that uses $K+1$ registers. The intervals in Y represent the live ranges of the variables in the loop. The control variables i and i_2 demand one register, which cannot be used in the allocation of the other live ranges inside the loop. Therefore, the coloring of $\mathcal{H}(Y,K)$ can be mapped trivially to the nodes of Y. □

Theorem 1. *The core register allocation problem for simple post-SSA programs is NP-complete.*

Proof. Combine Lemmas 4 and 5. □

As an illustrative example, to color C_5 with three colors is equivalent to determining a 3-coloring to the graph Y in Figure 7 (a). Such colorings can be found if and only if the core register allocation problem for $P = \mathcal{H}(Y, 3)$ can be solved with 4 registers. In this example, a solution exists. One assignment of registers would be $\{a, a_1, a_2, c, p_1\} \rightarrow R1$, $\{b, d, t_1, t_2, t, p_3\} \rightarrow R2$, $\{e, e_1, e_2, p2\} \rightarrow R3$, and $\{i, i_1, i_2, p_4\} \rightarrow R4$. This corresponds to coloring the arcs a and c with the first color, arcs b and d with the second, and e with the third.

4 Related Work

The first NP-completeness proof of a register allocation related problem was published by Sethi [22]. Sethi showed that, given a program represented as a set of instructions in a directed acyclic graph and an integer K, it is an NP-complete problem to determine if there is a computation of the DAG that uses at most K registers. Essentially, Sethi proved that the placement of loads and stores during the generation of code for a straight line program is an NP-complete problem if the order in which instructions appear in the target code is not fixed.

Much of the literature concerning complexity results for register allocation deals with two basic questions. The first is the core register allocation problem, which we defined in Section 1. The second is the core spilling problem which generalizes the core register allocation problem:

> **Core spilling problem:**
> Instance: a program P, number K of available registers, and a number M of temporaries.
> Problem: can at least M of the temporaries of P be mapped to one of the K registers such that temporary variables with interfering live ranges are assigned to different registers?

Farach and Liberatore [11] proved that the core spilling problem is NP-complete even for straight line code and even if rescheduling of instructions is not allowed. Their proof uses a reduction from set covering.

For a straight line program, the core register allocation problem is linear in the size of the interference graph. However, if the straight line program contains pre-colored registers that can appear more than once, then the core register allocation problem is NP-complete. In this case, register allocation is equivalent to pre-coloring extensions of interval graphs, which is NP-complete [2].

In the core register allocation problem, the number of registers K is not fixed. Indeed, the problem used in our reduction, namely the coloring of circular graphs, has a polynomial-time solution if the number of colors is fixed. Given n circular arcs determining a graph G, and a fixed number K of colors, Garey et al. [12] have given an $O(n \cdot K! \cdot K \cdot \log K)$ time algorithm for coloring G if such a coloring exists. Regarding general graphs, the coloring problem is NP-complete for every fixed value of $K > 2$ [13].

Bodlaender et al. [3] presented a linear-time algorithm for the core register allocation problem with a fixed number of registers for structured programs.

Their result holds even if rescheduling of instructions is allowed. If registers of different types are allowed, such as integer registers and floating point registers, for example, then the problem is no longer linear, although it is still polynomial.

Researchers have proposed different algorithms for inserting copy instructions, particularly for reducing the number of copy instructions [7, 5, 10, 18]. Rastello et al. [10] have proved that the optimum replacement of ϕ-functions by copy instructions is NP-complete. Their proof uses a reduction from the maximum independent set problem.

5 Conclusion

We have proved that the core register allocation problem is NP-complete for post-SSA programs generated by the classical approach to SSA-elimination that replaces ϕ-functions with copy instructions. In contrast, Hack et al.'s recent approach to SSA-elimination [17] generates programs for which the core register allocation problem is in polynomial time. We conclude that the choice of SSA-elimination algorithm matters.

We claim that compiler optimizations such as copy propagation and constant propagation cannot improve the complexity of the core register allocation problem for simple post-SSA programs. Inspecting the code in Figure 8 we perceive that the number of loop iterations cannot be easily predicted by a local analysis because all the control variables are given as function parameters. In the statement **int** v_j = i+$\mathcal{C}$; the variable i limits the effect of constant propagation and the use of different constants $\mathcal{C}$ limits the effect of copy propagation. Because all the $K + 1$ variables alive at the end of the loop have different values, live ranges cannot be merged at that point. In contrast, rescheduling of instructions might improve the complexity of the core register allocation problem for simple post-SSA programs. However, rescheduling combined with register allocation is an NP-complete problem even for straight line programs [22].

Theorem 1 continues to hold independent on the ordering in which copy instructions are inserted, because the function $\mathcal{G}$, defined in Section 2, can be modified to accommodate any ordering of the copy points. In more detail, let $W = \mathcal{F}(V, K)$ be a SSA-circular graph, let $n \in [0 \cdots \max(W)]$, and let $\mathrm{ovl}(n)$ be the number of intervals that overlap at point n.

$$\forall n \in \ ]max(V) \cdots max(W)] : \mathrm{ovl}(n) = K \qquad (7)$$

Any ordering that ensures property 7 suffices for the proof of Lemma 2. Figure 6 shows the region around the point 0 of a SSA-circular graph. Given $W = \mathcal{F}(V, K)$, exactly K copy points are inserted in the interval between $\max(V)$ and $\max(W)$.

Our proof is based on a reduction from the coloring of circular graphs. We proved our result for programs with a loop because the core of the interference graph of such programs is a circular graph. The existence of a back edge in the control flow graph is not a requirement for Theorem 1 to be true. For example, SSA-circular graphs can be obtained from a language with a single **if**-statement.

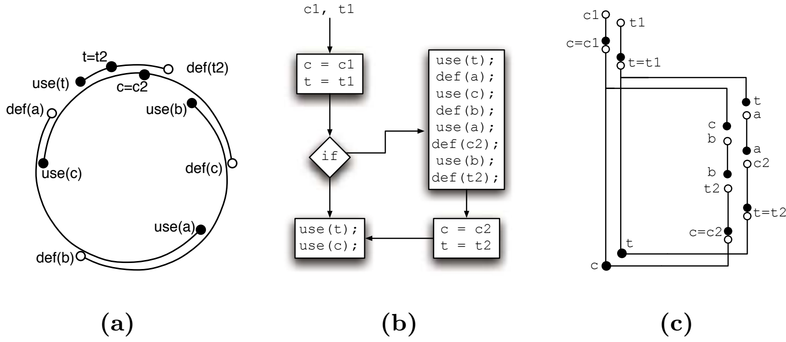


Fig. 9. (a) SSA graph W that represents $\mathcal{F}(C_3, K)$. (b) A program P that represents W with a single **if**-statement. (c) Schematic view of the live ranges of P.

Figure 9 (a) shows a SSA-circular graph that represents C_3, when $K = 2$, and Figure 9 (b) shows a program whose live ranges represent such graph. The live ranges are outlined in Figure 9 (c).

Acknowledgments. We thank Fabrice Rastello, Christian Grothoff and the anonymous referees for helpful comments on a draft of the paper. Fernando Pereira is sponsored by the Brazilian Ministry of Education under grant number 218603-9. Jens Palsberg is supported by the National Science Foundation award number 0401691.

References

1. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java.* Cambridge University Press, 2nd edition, 2002.
2. M. Biró, M. Hujter, and Zs. Tuza. Precoloring extension. I: interval graphs. In *Discrete Mathematics*, pages 267–279. ACM Press, 1992. Special volume (part 1) to mark the centennial of Julius Petersen's "Die theorie der regularen graphs".
3. Hans Bodlaender, Jens Gustedt, and Jan Arne Telle. Linear-time register allocation for a fixed number of registers. In *SIAM Symposium on Discrete Algorithms*, pages 574–583, 1998.
4. Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, 2005.
5. Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, 1998.
6. Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis.* ACM Press, 2005.
7. Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *International Conference on Programming Languages Design and Implementation*, pages 25–32. ACM Press, 2002.

8. Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
9. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
10. François de Ferriére, Christophe Guillon, and Fabrice Rastello. Optimizing the translation out-of-SSA with renaming constraints. *ST Journal of Research Processor Architecture and Compilation for Embedded Systems*, 1(2):81–96, 2004.
11. Martin Farach and Vincenzo Liberatore. On local register allocation. In *9th ACM-SIAM symposium on Discrete Algorithms*, pages 564 – 573. ACM Press, 1998.
12. M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Algebraic Discrete Methods*, 1(2): 216–227, 1980.
13. M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. *Theoretical Computer Science*, 1(3):193–267, 1976.
14. Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SICOMP*, 1(2):180–187, 1972.
15. Fanica Gavril. The intersection graphs of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatoric*, B(16):46–56, 1974.
16. Sebastian Hack. Interference graphs of programs in SSA-form. Technical Report ISSN 1432-7864, Universitat Karlsruhe, 2005.
17. Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *15th International Conference on Compiler Construction*. Springer-Verlag, 2006.
18. Allen Leung and Lal George. Static single assignment form for machine code. In *Conference on Programming Language Design and Implementation*, pages 204–214. ACM Press, 1999.
19. Daniel Marx. A short proof of the NP-completeness of circular arc coloring, 2003.
20. Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of APLAS'05, Asian Symposium on Programming Languages and Systems*, pages 315–329, 2005.
21. B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In *ACM SIGPLAN-SIGACT symposium on Principles of Programming languages*, pages 12–27. ACM Press, 1988.
22. Ravi Sethi. Complete register allocation problems. In *5th annual ACM symposium on Theory of computing*, pages 182–195. ACM Press, 1973.

A Logic of Reachable Patterns in Linked Data-Structures

Greta Yorsh[1,⋆], Alexander Rabinovich[1], Mooly Sagiv[1],
Antoine Meyer[2], and Ahmed Bouajjani[2]

[1] Tel Aviv Univ., Israel
{gretay, rabinoa, msagiv}@post.tau.ac.il
[2] Liafa, Univ. of Paris 7, France
{ameyer, abou}@liafa.jussieu.fr

Abstract. We define a new decidable logic for expressing and checking invariants of programs that manipulate dynamically-allocated objects via pointers and destructive pointer updates. The main feature of this logic is the ability to limit the neighborhood of a node that is reachable via a regular expression from a designated node. The logic is closed under boolean operations (entailment, negation) and has a finite model property. The key technical result is the proof of decidability.

We show how to express precondition, postconditions, and loop invariants for some interesting programs. It is also possible to express properties such as disjointness of data-structures, and low-level heap mutations. Moreover, our logic can express properties of arbitrary data-structures and of an arbitrary number of pointer fields. The latter provides a way to naturally specify postconditions that relate the fields on entry to a procedure to the fields on exit. Therefore, it is possible to use the logic to automatically prove partial correctness of programs performing low-level heap mutations.

1 Introduction

The automatic verification of programs with dynamic memory allocation and pointer manipulation is a challenging problem. In fact, due to dynamic memory allocation and destructive updates of pointer-valued fields, the program memory can be of arbitrary size and structure. This requires the ability to reason about a potentially infinite number of memory (graph) structures, even for programming languages that have good capabilities for data abstraction. Usually abstract-datatype operations are implemented using loops, procedure calls, and sequences of low-level pointer manipulations; consequently, it is hard to prove that a data-structure invariant is reestablished once a sequence of operations is finished [19].

To tackle the verification problem of such complex programs, several approaches emerged in the last few years with different expressive powers and levels of automation, including works based on abstract interpretation [27, 34, 31], logic-based reasoning [23, 32], and automata-based techniques [24, 28, 5]. An important issue is the definition of a formalism that (1) allows us to express relevant properties (invariants) of various kinds of linked data-structures, and (2) has the closure and decidability features needed

⋆ This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No 304/03).

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 94–110, 2006.
© Springer-Verlag Berlin Heidelberg 2006

for automated verification. The aim of this paper is to study such a formalism based on logics over arbitrary graph structures, and to find a balance between expressiveness, decidability and complexity.

Reachability is a crucial notion for reasoning about linked data-structures. For instance, to establish that a memory configuration contains no garbage elements, we must show that every element is reachable from some program variable. Other examples of properties that involve reachability are (1) the acyclicity of data-structure fragments, i.e., every element reachable from node u cannot reach u, (2) the property that a data-structure traversal terminates, e.g., there is a path from a node to a sink-node of the data-structure, (3) the property that, for programs with procedure calls when references are passed as arguments, elements that are *not* reachable from a formal parameter are not modified.

A natural formalism to specify properties involving reachability is the first-order logic over graph structures with transitive closure. Unfortunately, even simple decidable fragments of first-order logic become undecidable when transitive closure is added [13, 21].

In this paper, we propose a logic that can be seen as a fragment of the first-order logic with transitive closure. Our logic is (1) simple and natural to use, (2) expressive enough to cover important properties of a wide class of arbitrary linked data-structures, and (3) allows for algorithmic modular verification using programmer's specified loop-invariants and procedure's specifications.

Alternatively, our logic can be seen as a propositional logic with atomic proposition modelling reachability between heap objects pointed-to by program variables and other heap objects with certain properties. The properties are specified using patterns that limit the neighborhood of an object. For example, in a doubly linked list, a pattern says that if an object v has an an emanating `forward` pointer that leads to an object w, then w has a `backward` pointer into v.

The contributions of this paper can be summarized as follows:

– We define the *Logic of Reachable Patterns* (*LRP*) where reachability constraints such as those mentioned above can be used. Patterns in such constraints are defined by quantifier-free first-order formulas over graph structures and sets of access paths are defined by regular expressions.
– We show that *LRP* has a finite-model property, i.e., every satisfiable formula has a finite model. Therefore, invalid formulas are always falsified by a finite store.
– We prove that the logic *LRP* is, unfortunately, undecidable.
– We define a suitable restriction on the patterns leading to a fragment of *LRP* called LRP_2.
– We prove that the satisfiability (and validity) problem is decidable. The fragment LRP_2 is the main technical result of the paper and the decidability proof is non-trivial. The main idea is to show that every satisfiable LRP_2 formula is also satisfied by a tree-like graph. Thus, even though LRP_2 expresses properties of arbitrary data-structures, because the logic is limited enough, a formula that is satisfied on an arbitrary graph is also satisfied on a tree-like graph. Therefore, it is possible to answer satisfiability (and validity) queries for LRP_2 using a decision procedure for monadic second-order logic (MSO) on trees.

- We show that despite the restriction on patterns we introduce, the logic *LRP*$_2$ is still expressive enough for use in program verification: various important data-structures, and loop invariants concerning their manipulation, are in fact definable in *LRP*$_2$.

The new logic *LRP*$_2$ forms a basis of the verification framework for programs with pointer manipulation [37], which has important advantages w.r.t. existing ones. For instance, in contrast to decidable logics that restrict the graphs of interest (such as monadic second-order logic on trees), our logic allows arbitrary graphs with an arbitrary number of fields. We show that this is very useful even for verifying programs that manipulate singly-linked lists in order to express postcondition and loop invariants that relate the input and the output state. Moreover, our logic strictly generalizes the decidable logic in [3], which inspired our work. Therefore, it can be shown that certain heap abstractions including [16, 33] can be expressed using *LRP*$_2$ formulas.

The rest of the paper is organized as follows: Section 2 defines the syntax and the semantics of *LRP*, and shows that it has a finite model property, and that *LRP* is undecidable; Section 3 defines the fragment *LRP*$_2$, and demonstrates the expressiveness of *LRP*$_2$ on several examples; Section 4 describes the main ideas of the decidability proof for *LRP*$_2$; Section 5 discusses the limitations and the extensions of the new logics; finally, Section 6 discusses the related work. The full version of the paper [36] contains the formal definition of the semantics of *LRP* and proofs.

2 The *LRP* Logic

In this section, we define the syntax and the semantics of our logic. For simplicity, we explain the material in terms of expressing properties of heaps. However, our logic can actually model properties of arbitrary directed graphs. Still, the logic is powerful enough to express the property that a graph denotes a heap.

2.1 Syntax of *LRP*

LRP is a propositional logic over reachability constraints. That is, an *LRP* formula is a boolean combination of closed formulas in first-order logic with transitive closure that satisfy certain syntactic restrictions.

Let $\tau = \langle C, U, F \rangle$ denote a vocabulary, where (i) C is a finite set of constant symbols usually denoting designated objects in the heap, pointed to by program variables; (ii) U is a set of unary relation symbols denoting properties, e.g., color of a node in a Red-Black tree; (ii) F is a finite set of binary relation symbols (edges) usually denoting pointer fields.[1]

A **term** t is either a variable or a constant $c \in C$. An **atomic formula** is an equality $t = t'$, a unary relation $u(t)$, or an edge formula $t \xrightarrow{f} t'$, where $f \in F$, and t, t' are terms. A **quantifier-free formula** $\psi(v_0, \ldots, v_n)$ over τ and variables $v_0, \ldots, v_n$ is an arbitrary boolean combination of atomic formulas. Let $FV(\psi)$ denote the free variables of the formula ψ.

[1] We can also allow auxiliary constants and fields including abstract fields [8].

Definition 1. *Let ψ be a conjunction of edge formulas of the form $v_i \xrightarrow{f} v_j$, where $f \in F$ and $0 \leq i, j \leq n$. The **Gaifman graph** of ψ, denoted by B_ψ, is an undirected graph with a vertex for each free variable of ψ. There is an arc between the vertices corresponding to v_i and v_j in B_ψ if and only if $(v_i \xrightarrow{f} v_j)$ appears in ψ, for some $f \in F$. The **distance** between logical variables v_i and v_j in the formula ψ is the minimal edge distance between the corresponding vertices v_i and v_j in B_ψ.*

For example, for the formula $\psi = (v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2)$ the distance between v_1 and v_2 in ψ is 2, and its underlying graph B_ψ looks like this: $v_1 — v_0 — v_2$.

Definition 2. (Syntax of *LRP*) *A **neighborhood formula:** $N(v_0, \ldots, v_n)$ is a conjunction of edge formulas of the form $v_i \xrightarrow{f} v_j$, where $f \in F$ and $0 \leq i, j \leq n$.*

*A **routing expression** is an extended regular expression, defined as follows:*

$R ::= \emptyset$			*empty set*
$\mid$	ϵ		*empty path*
$\mid$	$\xrightarrow{f}$	$f \in F$	*forward along edge*
$\mid$	$\xleftarrow{f}$	$f \in F$	*backward along edge*
$\mid$	u	$u \in U$	*test if u holds*
$\mid$	$\neg u$	$u \in U$	*test if u does not hold*
$\mid$	c	$c \in C$	*test if c holds*
$\mid$	$\neg c$	$c \in C$	*test if c does not hold*
$\mid$	$R_1.R_2$		*concatenation*
$\mid$	$R_1 \vert R_2$		*union*
$\mid$	R^*		*Kleene star*

A routing expression can require that a path traverse some edges backwards. A routing expression has the ability to test presence and absence of certain unary relations and constants along the path.

*A **reachability constraint** is a closed formula of the form:*

$$\forall v_0, \ldots, v_n . R(c, v_0) \Rightarrow (N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n))$$

where $c \in C$ is a constant, R is a routing expression, N is a neighborhood formula, and ψ is an arbitrary quantifier-free formula, such that $FV(N) \subseteq \{v_0, \ldots, v_n\}$ and $FV(\psi) \subseteq FV(N) \cup \{v_0\}$. In particular, if the neighborhood formula N is $\mathtt{true}$ (the empty conjunction), then ψ is a formula with a single free variable v_0.

*An **LRP formula** is a boolean combination of reachability constraints.*

The subformula $N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$ defines a **pattern**, denoted by $p(v_0)$. Here, the designated variable v_0 denotes a "central" node of the "neighborhood" reachable from c by following an R-path. Intuitively, neighborhood formula N binds the variables $v_0, \ldots, v_n$ to nodes that form a subgraph, and ψ defines more constraints on those nodes. [2]

[2] In all our examples, a neighborhood formula N used in a pattern is such that B_N (the Gaifman graph of N) is connected.

We use **let** expressions to specify the scope in which the pattern is declared:

$$\textbf{let}\ p_1(v_0) \stackrel{\text{def}}{=} N_1(v_0, v_1, \ldots, v_n) \Rightarrow \psi_1(v_0, \ldots, v_n)\ \textbf{in}\ \varphi$$

This allows us to write more concise formulas via sharing of patterns.

Shorthands. We use $c[R]p$ to denote a reachability constraint. Intuitively, the reachability constraint requires that every node that is reachable from c by following an R-path satisfy the pattern p.

We use $c_1[R]\neg c_2$ to denote **let** $p(v_0) \stackrel{\text{def}}{=} (true \Rightarrow \neg(v_0 = c_2))$ **in** $c_1[R]p$. In this simple case, the neighborhood is only the node assigned to v_0. Intuitively, $c_1[R]\neg c_2$ means that the node labelled by constant c_2 is not reachable along an R-path from the node labelled by c_1. We use $c_1\langle R\rangle c_2$ as a shorthand for $\neg(c_1[R]\neg c_2)$. Intuitively, $c_1\langle R\rangle c_2$ means that *there exists* an R-path from c_1 to c_2. We use $c_1 = c_2$ to denote $c_1\langle\epsilon\rangle c_2$, and $c_1 \neq c_2$ to denote $\neg(c_1 = c_2)$. We use $c[R](p_1 \wedge p_2)$ to denote $(c[R]p_1) \wedge (c[R]p_2)$, when p_1 and p_2 agree on the central node variable. When two patterns are often used together, we introduce a name for their conjunction (instead of naming each one separately): **let** $p(v_0) \stackrel{\text{def}}{=} (N_1 \Rightarrow \psi_1) \wedge (N_2 \Rightarrow \psi_2)$ **in** φ.

In routing expressions, we use Σ to denote $(\stackrel{f_1}{\rightarrow} \mid \stackrel{f_2}{\rightarrow} \mid \ldots \mid \stackrel{f_m}{\rightarrow})$, the union of all the fields in F. For example, $c_1[\Sigma^*]\neg c_2$ means that c_2 is not reachable from c_1 by any path. Finally, we sometimes omit the concatenation operator "." in routing expressions.

Semantics. An interpretation for an *LRP* formula over $\tau = \langle C, U, F\rangle$ is a labelled directed graph $G = \langle V^G, E^G, C^G, U^G\rangle$ where: (i) V^G is a set of nodes modelling the heap objects, (ii) $E^G\colon F \to \mathcal{P}(V^G \times V^G)$ are labelled edges, (iii) $C^G\colon C \to V^G$ provides interpretation of constants as unique labels on the nodes of the graph, and (iv) $U^G\colon U \to \mathcal{P}(V^G)$ maps unary relation symbols to the set of nodes in which they hold.

We say that node $v \in G$ is labelled with σ if $\sigma \in C$ and $v = C^G(\sigma)$ or $\sigma \in U$ and $v \in U^G(\sigma)$. In the rest of the paper, *graph* denotes a directed labelled graph, in which nodes are labelled by constant and unary relation symbols, and edges are labelled by binary relation symbols, as defined above.

We define a satisfaction relation $\models$ between a graph G and *LRP* formula ($G \models \varphi$) similarly to the usual semantics the first-order logic with transitive closure over graphs (see [36]).

2.2 Properties of *LRP*

LRP with arbitrary patterns has a finite model property. If formula $\varphi \in$ *LRP* has an infinite model, each reachability constraint in φ that is satisfied by this model has a finite witness.

Theorem 1. (Finite Model Property): *Every satisfiable LRP formula is satisfiable by a finite graph.*

Sketch of Proof: We show that *LRP* can be translated into a fragment of an infinitary logic that has a finite model property. Observe that $c[R]p$ is equivalent to an infinite

conjunction of universal first-order sentences. Therefore, if G is a model of $c[R]p$ then every substructure of G is also its model. Dually, $\neg c[R]p$ is equivalent to an infinite disjunction of existential first-order sentences. Therefore, if G is a model of $\neg c[R]p$, then G has a finite substructure G' such that every substructure of G that contains G' is a model of $\neg c[R]p$. It follows that every satisfiable boolean combination of formulas of the form $c[R]p$ has a finite model. Thus, *LRP* has a finite model property.

The logic LRP is undecidable. The proof uses a reduction from the halting problem of a Turing machine.

Theorem 2. (Undecidability): *The satisfiability problem of LRP formulas is undecidable.*

Sketch of Proof: Given a Turing machine M, we construct a formula φ_M such that φ_M is satisfiable if and only if the execution of M eventually halts.

The idea is that each node in the graph that satisfies φ_M describes a cell of a tape in some configuration, with unary relation symbols encoding the symbol in each cell, the location of the head and the current state. The n-edges describe the sequence of cells in a configuration and a sequence of configurations. The b-edges describe how the cell is changed from one configuration to the next. The constant c_1 marks the node that describes the first cell of the tape in the first configuration, the constant c_2 marks the node that describes the first cell in the second configuration, and the constant c_3 marks the node that describes the last cell in the last configuration (see sketch in Fig. 1).

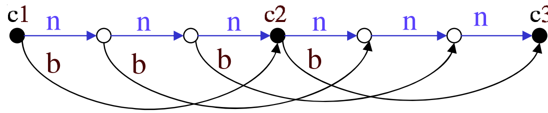


Fig. 1. Sketch of a model

The most interesting part of the formula φ_M ensures that all graphs that satisfy φ_M have a grid-like form. It states that for every node v that is n-reachable from c_1, if there is a b-edge from v to u, then there is a b-edge from the n-successor of v to the n-successor of u:

$$\textbf{let } p(v) \stackrel{\text{def}}{=} (v \xrightarrow{b} u) \wedge (v \xrightarrow{n} v_1) \wedge (u \xrightarrow{n} u_1) \Rightarrow (v_1 \xrightarrow{b} u_1) \textbf{ in } c_1[(\xrightarrow{n})^*]p \quad (1)$$

Remark. The reduction uses only two binary relation symbols and a fixed number of unary relation symbols. It can be modified to show that the logic with three binary relation symbols (and no unary relations) is undecidable.

3 The *LRP*$_2$ Fragment and Its Usefulness

In this section we define the *LRP*$_2$ fragment of *LRP*, by syntactically restricting the patterns. The main idea is to limit the distance between the nodes in the pattern in certain situations.

Definition 3. *A formula is in LRP*$_2$ *if in every reachability constraint* $c[R]p$*, with a pattern* $p(v_0) \stackrel{\text{def}}{=} N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n)$*,* ψ *has one of the following forms:*

- ***(equality pattern)*** ψ *is a an equality between variables* $v_i = v_j$*, where* $0 \leq i, j \leq n$*, and the distance between* v_i *and* v_j *in* N *is at most* 2 *(distance is defined in Def. 1),*
- ***(edge pattern)*** ψ *is of the form* $v_i \xrightarrow{f} v_j$ *where* $f \in F$ *and* $0 \leq i, j \leq n$*, and the distance between* v_i *and* v_j *in* N *is at most* 1.
- ***(negative pattern)*** *atomic formulas appear only negatively in* ψ*.*

Remark. Note that formula (1), which is used in the proof of undecidability in Theorem 2, is not in *LRP*$_2$, because p is an edge pattern with distance 3 between v_1 and u_1, while *LRP*$_2$ allows edge patterns with distance at most 1.

3.1 Describing Linked Data-Structures

In this section, we show that *LRP*$_2$ can express properties of data-structures. Table 1 lists some useful patterns and their meanings. For example, the first pattern det_f means that there is at most one outgoing f-edge from a node. Another important pattern uns_f means that a node has at most one incoming f-edge. We use the subscript f to emphasize that this definition is parametric in f.

Well-formed Heaps. We assume that C (the set of constant symbols) contains a constant for each pointer variable in the program (denoted by x, y in our examples). Also, C contains a designated constant $null$ that represents NULL values. Throughout the rest of the paper we assume that all the graphs denote well-formed heaps, i.e., the fields of all objects reachable from constants are deterministic, and dereferencing NULL yields $null$. In *LRP*$_2$ this is expressed by the formula:

$$(\bigwedge_{c \in C} \bigwedge_{f \in F} c[\Sigma^*]det_f) \wedge (\bigwedge_{f \in F} null\langle \xrightarrow{f} \rangle null) \tag{2}$$

Table 1. Useful pattern definitions ($f, b, g \in F$ are edge labels)

Pattern Name	**Pattern Definition**	**Meaning**
$det_f(v_0)$	$(v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2) \Rightarrow (v_1 = v_2)$	f-edge from v_0 is deterministic
$uns_f(v_0)$	$(v_1 \xrightarrow{f} v_0) \wedge (v_2 \xrightarrow{f} v_0) \Rightarrow (v_1 = v_2)$	v_0 is not heap-shared by f-edges
$uns_{f,g}(v_0)$	$(v_1 \xrightarrow{f} v_0) \wedge (v_2 \xrightarrow{g} v_0) \Rightarrow false$	v_0 is not heap-shared by f-edge and g-edge
$inv_{f,b}(v_0)$	$(v_0 \xrightarrow{f} v_1 \Rightarrow v_1 \xrightarrow{b} v_0)$ $\wedge\, (v_0 \xrightarrow{b} v_1 \Rightarrow v_1 \xrightarrow{f} v_0)$	edges f and b form a doubly-linked list between v_0 and v_1
$same_{f,g}(v_0)$	$(v_0 \xrightarrow{f} v_1 \Rightarrow v_0 \xrightarrow{g} v_1)$ $\wedge\, (v_0 \xrightarrow{g} v_1 \Rightarrow v_0 \xrightarrow{f} v_1)$	edges f and g emanating from v_0 are parallel

Using the patterns in Table 1, Table 2 defines some interesting properties of data-structures using *LRP*$_2$. The formula $reach_{x,f,y}$ means that the object pointed-to by the program variable y is reachable from the object pointed-to by the program variable x by following an access path of f field pointers. We can also use it with $null$ in the place of y. For example, the formula $reach_{x,f,null}$ describes a (possibly empty) linked-list pointed-to by x. Note that it implies that the list is acyclic, because $null$ is always a "sink" node in a well-formed heap. We can also express that there are no incoming f-edges into the list pointed to by x, by conjoining the previous formula with $unshared_{x,f}$. Alternatively, we can specify that x is located on a cycle of f-edges: $cyclic_{x,f}$. Disjointness can be expressed by the formula $disjoint_{x,f,y,g}$ that uses both forward and backward traversal of edges in the routing expression. For example, we can express that the linked list pointed to by x is disjoint from the linked-list pointed to by y, using the formula $disjoint_{x,f,y,f}$. Disjointness of data-structures is important for parallelization (e.g., see [17]).

Table 2. Properties of data-structures expressed in *LRP*$_2$

Name	**Formula**
$reach_{x,f,y}$	$x\langle(\xrightarrow{f})^*\rangle y$ the heap object pointed-to by y is reachable from the heap object pointed-to by x.
$cyclic_{x,f}$	$x\langle(\xrightarrow{f})^+\rangle x$ cyclicity: the heap object pointed-to by x is located on a cycle.
$unshared_{x,f}$	$x[(\xrightarrow{f})^*]uns_f$ every heap object reachable from x by an f-path has at most one incoming f-edge.
$disjoint_{x,f,y,g}$	$x[(\xrightarrow{f})^*(\xleftarrow{g})^*]\neg y$ disjointness: there is no heap object that is reachable from x by an f-path and also reachable from y by a g-path.
$same_{x,f,g}$	$x[(\xrightarrow{f} \mid \xrightarrow{g})*]same_{f,g}$ the f-path and the g-path from x are parallel, and traverse same objects.
$inverse_{x,f,b,y}$	$reach_{x,f,y} \wedge x[(\xrightarrow{f} .\neg y)^*]inv_{f,b}$ doubly-linked lists between two variables x and y with f and b as forward and backward edges.
$tree_{root,r,l}$	$root[(\xrightarrow{l} \mid \xrightarrow{r})^*](uns_{l,r} \wedge uns_l \wedge uns_r) \wedge \neg(root\langle(\xrightarrow{l} \mid \xrightarrow{r})^*\rangle root)$ tree rooted at $root$.

The last two examples in Table 2 specify data-structures with multiple fields. The formula $inverse_{x,f,b,y}$ describes a doubly-linked with variables x and y pointing to the head and the tail of the list, respectively. First, it guarantees the existence of an f-path. Next, it uses the pattern $inv_{f,b}$ to express that if there is an f-edge from one node to another, then there is a b-edge in the opposite direction. This pattern is applied to all nodes on the f-path that starts from x and that does not visit y, expressed using the test "$\neg y$" in the routing expression. The formula $tree_{root,r,l}$ describes a binary tree. The first part requires that the nodes reachable from the root (by following any path of l and r fields) be not heap-shared. The second part prevents edges from pointing back to the root of the tree by forbidding the root to participate in a cycle.

3.2 Expressing Verification Conditions

The `reverse` procedure shown in Fig. 2 performs in-place reversal of a singly-linked list. This procedure is interesting because it destructively updates the list and requires two fields to express partial correctness. Moreover, it manipulates linked lists in which each list node can be pointed-to from the outside. In this section, we show that the verification conditions for the procedure `reverse` can be expressed in LRP_2. If the verification conditions are valid, then the program is partially correct with respect to the specification. The validity of the verification conditions can be checked automatically because the logic LRP_2 is decidable, as shown in the next section. In [37], we show how to automatically generate verification conditions in LRP_2 for arbitrary procedures that are annotated with preconditions, postconditions, and loop invariants in LRP_2.

```
Node reverse(Node x){
  L0: Node y = NULL;
  L1: while (x != NULL){
  L2:     Node t = x->n;
  L3:     x->n = y;
  L4:     y = x;
  L5:     x = t;
  L6: }
  L7: return y;
}
```

Fig. 2. Reverse

Notice that in this section we assume that all graphs denote valid stores, i.e., satisfy (2). The precondition requires that x point to an acyclic list, on entry to the procedure. We use the symbols x^0 and n^0 to record the values of the variable x and the n-field on entry to the procedure.

$$pre \stackrel{\text{def}}{=} x^0 \langle (\stackrel{n^0}{\rightarrow})^* \rangle null^0$$

The postcondition ensures that the result is an acyclic list pointed-to by y. Most importantly, it ensures that each edge of the original list is reversed in the returned list, which is expressed in a similar way to a doubly-linked list, using $inverse$ formula. We use the relation symbols y^7 and n^7 to refer to the values on exit.

$$post \stackrel{\text{def}}{=} y^7 \langle (\stackrel{n^7}{\rightarrow})^* \rangle null^7 \wedge inverse_{x^0,n^0,n^7,y^7}$$

The loop invariant φ shown below relates the heap on entry to the procedure to the heap at the beginning of each loop iteration (label `L1`). First, we require that the part of the list reachable from x be the same as it was on entry to `reverse`. Second, the list reachable from y is reversed from its initial state. Finally, the only original edge outgoing of y is to x.

$$\varphi \stackrel{\text{def}}{=} same_{x^1,n^0,n^1} \wedge inverse_{x^0,n^0,n^1,y^1} \wedge x^0 \langle \stackrel{n^0}{\rightarrow} \rangle y^1$$

Note that the postcondition uses two binary relations, n^0 and n^7, and also the loop invariant uses two binary relations, n^0 and n^1. This illustrates that reasoning about singly-linked lists requires more than one binary relation.

The verification condition of `reverse` consists of two parts, VC_{loop} and VC, explained below.

The formula VC_{loop} expresses the fact that φ is indeed a loop invariant. To express it in our logic, we use several copies of the vocabulary, one for each program point. Different copies of the relation symbol n in the graph model values of the field n at different program points. Similarly, for constants. For example, Fig. 3 shows a graph that satisfies the formula VC_{loop} below. It models the heap at the end of some loop iteration of `reverse`. The superscripts of the symbol names denote the corresponding program points.

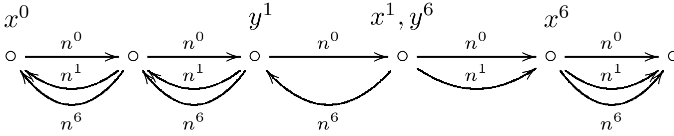


Fig. 3. An example graph that satisfies the VC_{loop} formula for `reverse`

To show that the loop invariant φ is maintained after executing the loop body, we assume that the loop condition and the loop invariant hold at the beginning of the iteration, and show that the loop body was executed without performing a null-dereference, and the loop invariant holds at the end of the loop body:

$$
\begin{array}{lll}
VC_{loop} \stackrel{\text{def}}{=} & (x \neq null) & \text{loop is entered} \\
& \wedge\varphi & \text{loop invariant holds on loop head} \\
& \wedge(y^6 = x^1) \wedge x^1\langle n^1\rangle x^6 \wedge x^1\langle n^6\rangle y^1 & \text{loop body} \\
& \wedge same_{y^1,n^1,n^6} \wedge same_{x^1,n^1,n^6} & \text{rest of the heap remains unchanged} \\
\Rightarrow & (x^1 \neq null) & \text{no null-derefernce in the body} \\
& \wedge\varphi^6 & \text{loop invariant after executing loop body}
\end{array}
$$

Here, φ^6 denotes the loop-invariant formula φ after executing the loop body (label `L6`), i.e., replacing all occurrences of x^1, y^1 and n^1 in φ by x^6, y^6 and n^6, respectively. The formula VC_{loop} defines a relation between three states: on entry to the procedure, at the beginning of a loop iteration and at the end of a loop iteration.

The formula VC expresses the fact that if the precondition holds and the execution reaches procedure's exit (i.e., the loop is not entered because the loop condition does not hold), the postcondition holds on exit: $VC \stackrel{\text{def}}{=} pre \wedge (x^1 = null) \Rightarrow post$.

4 Decidability of *LRP*$_2$

In this section, we show that *LRP*$_2$ is decidable for validity and satisfiability. Since *LRP*$_2$ is closed under negation, it is sufficient to show that it is decidable for satisfiability.

The satisfiability problem for *LRP*$_2$ is decidable. The proof proceeds as follows:

1. Every formula $\varphi \in$ *LRP*$_2$ can be translated into an equi-satisfiable normal-form formula that is a disjunction of formulas in $CLRP_2$ (Def. 4 and Theorem 3). It is sufficient to show that the satisfiability of $CLRP_2$ is decidable.
2. Define a class of simple graphs $\mathcal{A}_k$, for which the Gaifman graph is a tree with at most k additional edges (Def. 5).
3. Show that if formula $\varphi \in CLRP_2$ has a model, φ has a model in $\mathcal{A}_k$, where k is linear in the size of the formula φ (Theorem 4). This is the main part of the proof.
4. Translate formula $\varphi \in CLRP_2$ into an equivalent MSO formula.
5. Show that the satisfiability of MSO logic over $\mathcal{A}_k$ is decidable, by reduction to MSO on trees [30]. We could have also shown decidability using the fact that the tree width of all graphs in $\mathcal{A}_k$ is bounded by k, and that MSO over graphs with bounded tree width is decidable [11, 1, 35].

Definition 4. (Normal-Form Formulas): *A formula in $CLRP_2$ is a conjunction of reachability constraints of the form $c_1\langle R\rangle c_2$ and $c[R]p$, where p is one of the patterns allowed in LRP$_2$ (Def. 3). A normal-form formula is a disjunction of $CLRP_2$ formulas.*

Theorem 3. *There is a computable translation from LRP$_2$ to a disjunction of formulas in $CLRP_2$ that preserves satisfiability.*

Ayah Graphs. We define a notion of a simple tree-like directed graph, called Ayah graph.

Let $\mathcal{G}(S)$ denote the *Gaifman graph* of the graph S, i.e., an undirected graph obtained from S by removing node labels, edge labels, and edge directions (and parallel edges). The *distance* between nodes v_1 and v_2 in S is the number of edges on the shortest path between v_1 and v_2 in $\mathcal{G}(S)$. An undirected graph B is in T^k if removing self loops and at most k additional edges from B results in an acyclic graph.

Definition 5. *For $k \geq 0$, an Ayah graph of k is a graph S for which the Gaifman graph is in T^k: $\mathcal{A}_k = \{S | \mathcal{G}(S) \in T^k\}$.*

Let $\varphi \in CLRP_2$ be of the form $\varphi_\diamond \wedge \varphi_\Box \wedge \varphi_= \wedge \varphi_\rightarrow$, where $\varphi_\diamond$ is a conjunction of constraints of the form $c_1\langle R\rangle c_2$, $\varphi_\Box$ is a conjunction of reachability constraints with negative patterns, $\varphi_=$ is a conjunction of reachability constraints with equality patterns, and $\varphi_\rightarrow$ is a conjunction of reachability constraints with edge patterns.

Theorem 4. *If $\varphi \in CLRP_2$ is satisfiable, then φ is satisfiable by a graph in $\mathcal{A}_k$, where $k = 2 \times n \times |C| \times m$, m is the number of constraints in $\varphi_\diamond$, $|C|$ is the number of constants in the vocabulary, and for every regular expression that appears in $\varphi_\diamond$ there is an equivalent automaton with at most n states.*

Sketch of Proof: Let S be a model of $\varphi : S \models \varphi$. We construct a graph S' from S and show that $S' \models \varphi$ and $S' \in \mathcal{A}_k$. The construction uses the following operations on graphs.

Witness Splitting. A witness W for a formula $c_1\langle R\rangle c_2$ in *CLRP*$_2$ in a graph S is a path in S, labelled with a word $w \in L(R)$, from the node labelled with c_1 to the node labelled with c_2. Note that the nodes and edges on a witness path for R need not be distinct. Using W, we construct a graph W' that consists of a path, labelled with w, that starts at the node labelled by c_1 and ends at the node labelled by c_2. Intuitively, we duplicate a node of W each time the witness path for R traverses it, unless the node is marked with a constant. As a result, all shared nodes in W' are labelled with constants. Also, every cycle contains a node labelled with a constant. By construction, we get that $W' \models c_1\langle R\rangle c_2$. We say that W' is the result of *splitting* the witness W.

Finally, we say that W is the *shortest witness* for $c_1\langle R\rangle c_2$ if any other witness path for $c_1\langle R\rangle c_2$ is at least as long as W. The result of splitting the shortest witness is a graph in $\mathcal{A}_k$, where $k = 2 \times n \times |C|$: to break all cycles it is sufficient to remove all the edges adjacent to nodes labelled with constants, and a node labelled with a constant is visited at most n times. (If a node is visited more than once in the same state of the automaton, the path can be shortened.)

Merge Operation. Merging two nodes in a graph is defined in the usual way by gluing these nodes. Let $p(v_0) \stackrel{\text{def}}{=} N(v_0, v_1, v_2) \Rightarrow (v_1 = v_2)$ be an equality pattern. If a graph violates a reachability constraint c[R]p, we can assign nodes n_0, n_1, and n_2 to v_0, v_1, and v_2, respectively, such that there is a R-path from c to v_0, $N(n_0, n_1, n_2)$ holds, and n_1 and n_2 are distinct nodes. In this case, we say that *merge operation of* n_1 *and* n_2 *is enabled* (by c[R]p). The nodes n_1 and n_2 can be merge to discharge this assignment (other merge operations might still be enabled after merging n_1 and n_2).

Edge-Addition Operation. Let $p(v_0) \stackrel{\text{def}}{=} N(v_0, v_1, v_2) \Rightarrow v_1 \xrightarrow{f} v_2$ be an edge pattern. If a graph violates a reachability constraint c[R]p, we can assign nodes n_0, n_1, and n_2 to v_0, v_1, and v_2, respectively, such that there is a R-path from c to v_0, $N(n_0, n_1, n_2)$ holds, and there is no f-edge from n_1 to n_2. In this case, we say that *edge-operation operation is enabled* (by c[R]p). We can add an f-edge from n_1 and n_2 to discharge this assignment.

The following lemma is the key observation of this proof.

Lemma 1. *The class of $\mathcal{A}_k$ graphs is closed under merge operations of nodes in distance at most two and edge-addition operations at distance one.*

Sketch of Proof: If an edge is added in parallel to an existing one (distance one), it does not affect the Gaifman graph, thus $\mathcal{A}_k$ is closed under edge-addition. The proof that $\mathcal{A}_k$ is closed under merge operations is more subtle [36].

In particular, the class $\mathcal{A}_k$ is closed under the merge and edge-addition operations forced by *LRP*$_2$ formulas. This is the only place in our proof where we use the distance restriction of *LRP*$_2$ patterns.

Given a graph S that satisfies φ, we construct the graph S' as follows:

1. For each constraint i in $\varphi_\diamond$, identify the shortest witness W_i in S. Let W_i' be the result of splitting the witness W_i.
2. The graph S_0 is a union of all W_i''s, in which the nodes labelled with the (syntactically) same constants are merged.

3. Apply all enabled merge operations and all enabled edge-addition operations in any order, producing a sequence of distinct graphs $S_0, S_1, \ldots, S_r$, until S_m has no enabled operations.
4. The result $S' = S_r$.

The process described above terminates after a finite number of steps, because in each step either the number of nodes in the graph is decreased (by merge operations) or the number of edges is increased (by edge-addition operations).

The proof proceeds by induction on the process described above. Initially, S_0 is in $\mathcal{A}_k$. By Lemma 1, all S_i created in the third step of the construction above are in $\mathcal{A}_k$; in particular, $S' \in \mathcal{A}_k$.

By construction of S_0, it contains a witness for each constraint in $\varphi_\Diamond$, and merge and edge-addition operations preserve the witnesses, thus S' satisfies $\varphi_\Diamond$. Moreover, S_0 satisfies all constraints in $\varphi_\Box$. We show that merge and edge-addition operations applied in the construction preserve $\varphi_\Box$ constraints, thus S' satisfies $\varphi_\Box$. The process above terminates when no merge and edge-addition operations are enabled, that is, S' satisfies $\varphi_= \wedge \varphi_\rightarrow$. Thus, S' satisfies φ.

The full proof is available at [36].

4.1 Complexity

We proved decidability by reduction to MSO on trees, which allows us to decide *LRP*$_2$ formulas using MONA decision procedure [18]. Alternatively, a decision procedure for *LRP*$_2$ can directly construct a tree automaton from a normal-form formula, and can then check emptiness of the automaton. The worst case complexity of the satisfiability problem of *LRP*$_2$ formulas is at least doubly-exponential, but it remains elementary (in contrast to MSO on trees, which is non-elementary); we are investigating tighter upper and lower bounds. The complexity depends on the bound k of $\mathcal{A}_k$ models, according to Theorem 4. If the routing expressions do not contain constant symbols, then the bound k does not depend on the routing expressions: it depends only on the number of reachability constraints of the form $c_1\langle R\rangle c_2$. The *LRP*$_2$ formulas that come up in practice are well-structured, and we hope to achieve a reasonable performance.

5 Limitations and Further Extensions

Despite the fact that *LRP*$_2$ is useful, there are interesting program properties that cannot be expressed. For example, transitivity of a binary relation, that can be used, e.g., to express partial orders, is naturally expressible in *LRP*, but not in *LRP*$_2$. Also, the property that a general graph is a tree in which each node has a pointer back to the root is expressible in *LRP*, but not in *LRP*$_2$. Notice that the property is non-trivial because we are operating on general graphs, and not just trees. Operating on general graphs allows us to verify that the data-structure invariant is reestablished after a sequence of low-level mutations that temporarily violate the invariant data-structure.

There are of course interesting properties that are beyond *LRP*, such as the property that a general graph is a tree in which every leaf has a pointer to the root of a tree.

In the future, we plan to generalize *LRP*$_2$ while maintaining decidability, perhaps beyond *LRP*. We are encouraged by the fact that the proof of decidability in Section 4 holds "as is" for many useful extensions. For example, we can generalize the patterns to allow neighborhood formulas with disjunctions and negations of unary relations. In fact, more complex patterns can be used, as long as they do not violate the $\mathcal{A}_k$ property. For example, we can define trees rooted at x with parent pointer b from every tree node to its parent by $tree_{x,r,l,b} \wedge$ **let** $p(v_0) \stackrel{\text{def}}{=} ((v_1 \stackrel{l}{\rightarrow} v_0) \vee (v_1 \stackrel{r}{\rightarrow} v_0)) \Rightarrow (v_0 \stackrel{b}{\rightarrow} v_1)$**in** $x[(\stackrel{l}{\rightarrow} \mid \stackrel{r}{\rightarrow})^*](det_b \wedge p)$. The extended logic remains decidable, because the pattern p adds edges only in parallel to the existing ones.

Currently, reachability constraints describe paths that start from nodes labelled by constants. We can show that the logic remains decidable when reachability constraints are generalized to describe paths that start from any node that satisfies a quantifier-free *positive* formula θ: $\forall v, w_0, \ldots, w_m, v_0, \ldots, v_n. R(v, v_0) \wedge \theta(v, w_0, \ldots, w_m) \Rightarrow (N(v_0, \ldots, v_n) \Rightarrow \psi(v_0, \ldots, v_n))$.

6 Related Work

There are several works on logic-based frameworks for reasoning about graph/heap structures. We mention here the ones which are, as far as we know, the closest to ours.

The logic *LRP* can be seen as a fragment of the first-order logic over graph structures with transitive closure (TC logic [20]). It is well known that TC is undecidable, and that this fact holds even when transitive closure is added to simple fragments of FO such as the decidable fragment L^2 of formulas with two variables [29, 15, 13].

It can be seen that our logics *LRP* and *LRP*$_2$ are both uncomparable with L^2 + TC. Indeed, in *LRP* no alternation between universal and existential quantification is allowed. On the other hand, *LRP*$_2$ allows us to express patterns (e.g., heap sharing) that require more than two variables (see Table 1, Section 3).

In [3], decidable logic L_r (which can also be seen as a fragment of TC) is introduced. The logics *LRP* and *LRP*$_2$ generalize L_r, which is in fact the fragment of these logics where only two fixed patterns are allowed: equality to a program variable and heap sharing.

In [21, 2, 26, 4] other decidable logics are defined, but their expressive power is rather limited w.r.t. *LRP*$_2$ since they allow at most one binary relation symbol (modelling linked data-structures with 1-selector). For instance, the logic of [21] does not allow us to express the reversal of a list. Concerning the class of 1-selector linked data-structures, [6] provides a decision procedure for a logic with reachability constraints and arithmetical constraints on lengths of segments in the structure. It is not clear how the proposed techniques can be generalized to larger classes of graphs. Other decidable logics [7, 25] are restricted in the sharing patterns and the reachability they can describe.

Other works in the literature consider extensions of the first-order logic with fixpoint operators. Such an extension is again undecidable in general but the introduction of the notion of (loosely) guarded quantification allows one to obtain decidable fragments such as μGF (or μLGF) (Guarded Fragment with least and greater fixpoint operators) [14, 12]. Similarly to our logics, the logic μGF (and also μLGF) has the tree model property: every satisfiable formula has a model of bounded tree width. However,

guarded fixpoint logics are incomparable with *LRP* and LRP_2. For instance, the LRP_2 pattern det_f that requires determinism of f-field, is not a (loosely) guarded formula.

The PALE system [28] uses an extension of the monadic second order logic on trees as a specification language. The considered linked data structures are those that can be defined as *graph types* [24]. Basically, they are graphs that can be defined as trees augmented by a set of edges defined using routing expressions (regular expressions) defining paths in the (undirected structure of the) tree. LRP_2 allows us to reason naturally about arbitrary graphs without limitation to tree-like structures. Moreover, as we show in Section 3, our logical framework allows us to express postconditions and loop invariants that relate the input and the output state. For instance, even in the case of singly-linked lists, our framework allows us to express properties that cannot be expressed in the PALE framework: in the list reversal example of Section 3, we show that the output list is precisely the reversed input list, whereas in the PALE approach, one can only establish that the output is a list that is the permutation of the input list.

In [22], we tried to employ a decision procedure for MSO on trees to reason about reachability. However, this places a heavy burden on the specifier to prove that the data-structures in the program can be simulated using trees. The current paper eliminated this burden by defining syntactic restrictions on the formulas and showing a general reduction theorem.

Other approaches in the literature use undecidable formalisms such as [17], which provides a natural and expressive language, but does not allow for automatic property checking.

Separation logic has been introduced recently as a formalism for reasoning about heap structures [32]. The general logic is undecidable [10] but there are few works showing decidable fragments [10, 4]. One of the fragments is propositional separation logic where quantification is forbidden [10, 9] and therefore seems to be incomparable with our logic. The fragment defined in [4] allows one to reason only about singly-linked lists with explicit sharing. In fact, the fragment considered in [4] can be translated to LRP_2, and therefore, entailment problems as stated in [4] can be solved as implication problems in LRP_2.

7 Conclusions

Defining decidable fragments of first order logic with transitive closure that are useful for program verification is a difficult task (e.g., [21]). In this paper, we demonstrated that this is possible by combining three principles: (i) allowing arbitrary boolean combinations of the reachability constraints, which are closed formulas without quantifier alternations, (ii) defining reachability using regular expressions denoting pointer access paths (not) reaching a certain pattern, and (iii) syntactically limiting the way patterns are formed. Extensions of the patterns that allow larger distances between nodes in the pattern either break our proof of decidability or are directly undecidable.

The decidability result presented in this paper improves the state-of-the-art significantly. In contrast to [21, 2, 26, 4], *LRP* allows several binary relations. This provides a natural way to (i) specify invariants for data-structures with multiple fields (e.g., trees, doubly-linked lists), (ii) specify post-condition for procedures that mutate pointer fields

of data-structures, by expressing the relationships between fields before and after the procedure (e.g., list reversal, which is beyond the scope of PALE), (iii) express verification conditions using a copy of the vocabulary for each program location.

We are encouraged by the expressiveness of this simple logic and plan to explore its usage for program verification and abstract interpretation.

References

1. S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, 1991.
2. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI*, pages 164–180, 2005.
3. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symp. On Programming*, pages 2–19, March 1999.
4. J. Berdine, C. Calcagno, and P. O'Hearn. A Decidable Fragment of Separation Logic. In *FSTTCS'04*. LNCS 3328, 2004.
5. A. Bouajjani, P. Habermehl, P.Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS '05*, volume 3440 of *LNCS*. Springer, 2005.
6. M. Bozga and R. Iosif. Quantitative Verification of Programs with Lists. In *VISSAS intern. workshop*. IOS Press, 2005.
7. M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In *Static Analysis Symp.*, pages 344–360, 2004.
8. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Int. J. on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
9. C. Calcagno, P. Gardner, and M. Hague. From Separation Logic to First-Order Logic. In *FOSSACS'05*. LNCS 3441, 2005.
10. C. Calcagno, H. Yang, and P. O'Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *FSTTCS'01*. LNCS 2245, 2001.
11. B. Courcelle. The monadic second-order logic of graphs, ii: Infinite graphs of bounded width. *Mathematical Systems Theory*, 21(4):187–221, 1989.
12. E. Grädel. Guarded fixed point logic and the monadic theory of trees. *Theoretical Computer Science*, 288:129–152, 2002.
13. E. Grädel, M.Otto, and E.Rosen. Undecidability results on two-variable logics. *Archive of Math. Logic*, 38:313–354, 1999.
14. E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *LICS'99*. IEEE, 1999.
15. E. Graedel, P. Kolaitis, and M. Vardi. On the decision problem for two variable logic. *Bulletin of Symbolic Logic*, 1997.
16. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
17. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.
18. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, 1995.
19. C.A.R. Hoare. Recursive data structures. *Int. J. of Comp. and Inf. Sci.*, 4(2):105–132, 1975.
20. N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

21. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundery between decidability and undecidability of transitive closure logics. In *CSL*, 2004.
22. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In *CAV*, 2004.
23. S. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.
24. N. Klarlund and M. I. Schwartzbach. Graph Types. In *POPL'93*. ACM, 1993.
25. V. Kuncak and M. Rinard. Generalized records and spatial conjunction in role logic. In *Static Analysis Symp.*, Verona, Italy, August 26–28 2004.
26. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Symp. on Princ. of Prog. Lang.*, 2006. To appear.
27. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.
28. A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 221–231, 2001.
29. M. Mortimer. On languages with two variables. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:135–140, 1975.
30. M. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.
31. T. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *CAV*, pages 15–30, 2004.
32. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS'02*. IEEE, 2002.
33. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1): 1–50, January 1998.
34. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.
35. D. Seese. Interpretability and tree automata: A simple way to solve algorithmic problems on graphs closely related to trees. In *Tree Automata and Languages*, pages 83–114. 1992.
36. G. Yorsh, M. Sagiv, A. Rabinovich, A. Bouajjani, and A. Meyer. A logic of reachable patterns in linked data-structures. Technical report, Tel Aviv University, 2005. Available at "www.cs.tau.ac.il/∼gretay".
37. G. Yorsh, M. Sagiv, A. Rabinovich, A. Bouajjani, and A. Meyer. Verification framework based on the logic of reachable patterns. In preparation, 2005.

Dynamic Policy Discovery with Remote Attestation
(Extended Abstract)

Corin Pitcher* and James Riely**

CTI, DePaul University
{cpitcher, jriely}@cs.depaul.edu

Abstract. Remote attestation allows programs running on trusted hardware to prove their identity (and that of their environment) to programs on other hosts. Remote attestation can be used to address security concerns if programs agree on the meaning of data in attestations. This paper studies the enforcement of code-identity based access control policies in a hostile distributed environment, using a combination of remote attestation, dynamic types, and typechecking. This ensures that programs agree on the meaning of data and cannot violate the access control policy, even in the presence of opponent processes. The formal setting is a π-calculus with secure channels, process identity, and remote attestation. Our approach allows executables to be typechecked and deployed independently, without the need for secure initial key and policy distribution beyond the trusted hardware itself.

Keywords: remote attestation, code-identity based access control, policy establishment, key establishment, π-calculus, Next Generation Secure Computing Base.

1 Introduction

Processes in a distributed system often rely upon the trustworthiness of processes running on other hosts. The remote attestation mechanism in Microsoft's Next Generation Secure Computing Base (NGSCB) [39], in conjunction with trusted hardware specified by the Trusted Computing Group [50, 42], allows processes running on trusted hardware to attach evidence of their identity (and the identity of their environment) to data. Other processes can examine this evidence to assess the degree of trust to place in the process that attested to the data.

Enforcement of access control policies in hostile distributed environments has been a driving concern in the development of trusted hardware and remote attestation. We formalize these notions in a variant of the π-calculus [40], dubbed π-rat, and develop a type system that enforces access control policies in the presence of arbitrary opponents. The type system allows programs to be certified independently and deployed without shared keys or policies beyond those in the trusted hardware.

Organization. In the remainder of this introduction, we set out our goals and assumptions. Section 2 demonstrates the use of π-rat through an extended example. Section 3

* Supported by the DePaul University Research Council.
** Supported by the National Science Foundation under Grant No. 0347542.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 111–125, 2006.
© Springer-Verlag Berlin Heidelberg 2006

presents the dynamics of the language and formally defines runtime errors and robust safety. Section 4 develops a type system that ensures safety in the presence of arbitrary attackers, and sketches the proof of robust safety. We conclude with a discussion of related and future work.

Identity and Attestation. We assume that processes can be given *identities* in a uniform manner. We write $h\,[P]$ for a process running with identity h. Initial processes have the form $\#P\,[P]$, where # is a globally agreed hashing function on process terms. While a process may evolve, its identity cannot. Further, identities cannot be forged. Thus a process Q running with identity $\#P$, must be a residual of P. Our treatment of identities is deliberately abstract; our formal results do not use hash functions. Nonetheless, we write $\#P$ in examples to indicate the identity of a named process. We leave higher-order extensions to our language, internalizing # as an operator, to future work.

In this paper, we do not deal with other forms of identity. For example, there is no notion of code running on behalf of a principal [8], nor is there a notion of explicit distribution [28]. We assume that all resources are globally accessible, for example in a shared heap. Opponents are modeled as processes with access to these globally known resources. As a result, we make no distinction between the many instances of a program; thus $h\,[P] \mid h\,[Q]$ is indistinguishable from $h\,[P \mid Q]$.

At first approximation, an attestation to data M is a signature (with message recovery) upon the pair (h,M), where h is the hash of the process that requested the attestation. The signature is created by trusted hardware using its own private key. The hardware manufacturer issues a certificate for the trusted hardware's public key at the time of manufacture and stores it in the trusted hardware. Upon receipt of an attestation, and the certificate, the relying party verifies the certificate using the manufacturer's well-known public key, and then verifies the signature using the trusted hardware's public key. If successful, the relying party concludes that a process with hash h did request an attestation for M from the trusted hardware. To deduce further properties about M, the relying party must know more about the conditions under which the process with hash h is willing to attest to data.

Policies and Certification. We are interested in the distinction between processes that have been certified to obey a certain policy and those that have not been so certified. Realistically, one would like to model multiple kinds of policies and multiple methods of certification; however, here we limit attention to a single, extra-lingual certification, defined as a typing system. We encode policies in types, T, and allow for communication of policy between processes. The particular policies in this paper are access control policies based on code identity. For example, our system allows expression of policies such as "only the ACME media player may display this data".

Opponent processes are those which are not certified. Opponents cannot persuade trusted hardware to create false attestations, but otherwise their behavior is entirely unconstrained. We assume that a conservative approximation of the set of certified processes is available at runtime. That is, a process may inquire, at runtime, whether the process corresponding to a certain identity has been certified. To keep the language simple, we do not deal explicitly with distribution of certifications. In addition, our policies are stated directly in terms of program identities, rather than

allowing additional levels of indirection. Both limitations may be alleviated by incorporating a trust management framework, the investigation of which we leave to future work.

Unlike previous analyses of cryptographic protocols [9, 3, 22, 23, 25] remote attestation is intended to be used to establish secure channels starting from insecure channels that are accessible to opponents. In order to allow the communication of policy information during secure-channel establishment, we employ a form of dynamic typing [4, 33]. Our interpretation of $\mathsf{at}(h, M)$ is that h vouches for the policy encoded in M. In particular, if M is $\{N:T\}$ with asserted type T, then h vouches that N can safely be used with policy T.

The policy information in attestations from uncertified processes cannot be trusted. While the payload of such an attestation may be stored and communicated, it cannot safely be used in any other way.

Channels and Access Control. As usual in π, we encode data using channels. Thus access policies regulate the readers and writers of channels. Our policies do not limit possession of channels, only their use; although in the case of an opponent, possession and use are indistinguishable since opponents are not constrained to obey any policy. The policy for a channel is defined when the channel is created and may be communicated over insecure channels via attestations.

In keeping with our high-level interpretation of attestations, we avoid explicit cryptographic primitives [9]. In their place we adopt a polarized variant of the π-calculus [41] which allows transmission of read and write capabilities individually. This simplification is justified by Abadi, Fournet and Gonthier's work on implementing secure channel abstractions [8, 7].

Contributions. In terms of security policies, our aims are modest relative to other recent work on types in process languages. For example, we do not attempt to establish freshness [23, 25] or information flow [30] properties. Nonetheless, we achieve a concise statement of secrecy properties (cf. [1, 3, 17]). For example, if a value is created with type $\mathsf{Data}(h)$, then our typing system ensures that only the program with identity h can display it. Unusually for systems allowing arbitrary opponents [1, 3, 17, 23, 25], our typing system also ensures *memory safety* for certified processes; our approach to opponent typability is reminiscent of [43].

A distinctive aspect of our approach, in keeping with proposed applications of remote attestation, is to minimize reliance upon an authority to distribute keys and policies. For example, in media distribution systems, the executables share nothing but data that is both *public* (can be intercepted by the opponent) and *tainted* (may have come from the opponent). In contrast, spi processes typically require further agreement. Consider trustworthy spi processes P and Q deployed by a mutually-trusted authority that initializes the system "$\mathsf{new}\ kp;\ (P \mid Q)$." The authority may, for example, create the keypair kp and distribute the public key to P and the private key to Q. A common type environment ensures that P and Q agree on the meaning of data encrypted by kp.

2 Example

A prototypical use of remote attestation is to establish a channel for sending secrets to an instance of a trusted executable, such as a media player that enforces a favored access control policy. A process player on trusted hardware creates a fresh keypair, attests to the public key, then transmits the result to a server. The server verifies the attestation, concluding that the public key belongs to an instance of the player executable running on trusted hardware. Since the server trusts the player, it encrypts the data, perhaps a movie, and sends the ciphertext back to player. The player is relied upon to enforce a policy, such as not making the data available to other processes, or limiting the number of viewings. The trusted hardware hosting player is relied upon to prevent anyone, including the host's administrator, from violating the player's environment.

MEDIA PLAYER EXAMPLE

```
PWr ≜ Wr⟨any,#player⟩ (Tnt)
player ≜
(1) new pch:Ch⟨any,#player⟩ (Tnt) ;
(2) wr (sch) !at (#player, {wr (pch) :PWr}) ;
(3) rd (pch) ?smsg;
(4) let at (s, mdyn) =smsg;
(5) iscert s;
(6) typecase {m:PData} =mdyn;
(7) display m

PData ≜ Data (#player)
server ≜
 (8) repeat rd (sch) ?pmsg;
 (9) let at (p, chdyn) =pmsg;
(10) iscert p;
(11) typecase {wr:PWr} =chdyn;
(12) new n:PData;
(13) wr!at (#server, {n:PData})
```

We formalize this example at the top of the page. Initially, the player and server agree only on the name of an untrusted (available to the opponent) channel sch, which has type Ch⟨any,any⟩ (Un) ; the angled brackets contain the channel's policy, the parentheses contain the type of values communicated. The type of sch indicates that anyone (including opponents) may send or receive messages on the channel and that the values communicated are untrusted. The player and server must also have compatible policies for the write capability (representing one key from a keypair) and data, with names PWr and PData respectively. The policies mention the hash of the player program, and thus the two.

The player (1) creates a channel of type Ch⟨any,#player⟩ (Tnt) (representing a keypair), and (2) communicates the write capability (one of the keys) of type PWr to the server by writing on sch. The access control policy associated with the channel pch is ⟨any,#player⟩. The first component any indicates that any executable, certified (typed) or not, may write to the channel; thus the received value is tainted. Using the more restrictive #server as the first component of the policy, meaning that only the server may write to the channel, could be violated after the write capability is communicated on the insecure channel sch. The second component of the policy #player means that only an instance of the player executable can read from the channel.

A more lenient access control policy ⟨any,cert⟩ for pch would allow any well-typed executable, denoted cert, to read from the channel. These two policies illustrate the

difference between possession and use in π-rat, because any well-typed executable can possess the read capability for pch—regardless of whether the access control policy is $\langle \mathsf{any}, \#\mathsf{player} \rangle$ or $\langle \mathsf{any}, \mathsf{cert} \rangle$. Both cases are safe because well-typed executables will only use the read capability when they are certain that it is permitted by the access control policy specified by the channel's creator.

The media server (8) repeatedly reads sch. Upon receipt of a message, the server (9) unpacks the attestation in the message, discovering the hash of the attesting process, (10) checks that the hash is certified (the hash of a well-typed executable), then (11) unpacks the payload of the message (the write capability) which involves checking that the stated policy complies with the expected policy. The server then (12) creates a data object and (13) sends it to the player via the write capability. In a similar fashion, (3)-(6) the player receives the data and verifies its origin and policy, then (7) displays the data.

Lines (2) and (13) include attestations. Remote attestation does not allow a remote process to force trusted hardware to identify an uncooperative process. However, processes that are unwilling to identify themselves using attestation may find other processes unwilling to interact with them.

From an implementation perspective, using a hash other than the hash of the enclosing process as the first component of the attestation primitive is unimplementable because trusted hardware will only create attestations with the hash of the requesting process. Due to inherent circularity, it is impossible for an executable to contain its own hash, so we assume that a process is able to query the trusted hardware to find its own hash at runtime: in which case a typechecker implementation would need to verify that the code to perform the query is correct. A more interesting challenge for distributed systems using remote attestation is that two executables cannot contain each other's hashes—one executable may contain the hash of the other executable, as illustrated by the media server code which can only be written after the hash of the player executable is known. Of course, two processes may learn one another's hashes, and incorporate those hashes into policies, during the course of execution.

The media server generates the data with a policy stating that it is only usable by the player. The data stored in n is sent to the player using the write capability wr(pch), so no-one but the player can receive the message. The data is sent inside an attestation, because the player has no reason to trust data that it receives on pch. The type inside the attestation is checked by the player to ensure that it treats the data in accordance with the system's access control policy. When the player receives the hash, it must dynamically check that the hash is that of a well-typed executable. This is necessary to ensure that the type in the attestation is reliable.

The threat considered here is that a process will read or write to a channel in violation of the policy of a well-typed executable that created the channel. For example, we would like to prevent an executable other than player from displaying data n. Our main theorem states that access control violations cannot occur in well-typed configurations, even if the well-typed configuration is placed in parallel with an untyped opponent.

3 Dynamics

We give the syntax and dynamic semantics of π-rat. We describe runtime errors and define safety. We describe types, T, in the next section.

Syntax and Evaluation. The language has syntactic categories for names, terms, processes and configurations. Evaluation is defined in terms of configurations. Assuming a non-colliding hash function (#) on programs — such that if $\#P = \#Q$ then $P = Q$ — initial configurations have the following form.

$$(\#P_1)\,[P_1] \mid \cdots \mid (\#P_m)\,[P_m]$$

The configuration represents m concurrent processes, each identified by its hash. This form is not preserved by reduction, since a process may evolve, but its hash does not. (In practice, remote attestation uses the hash of the executable, and the remaining state of the process is ignored.) We thus choose to treat hashes abstractly as names.

Names (*a-z*) serve several purposes. To aid the reader, we use x, y, z to stand for variables, h, g, f to stand for hashes or hash-typed variables, and a, b, c to stand for channels.

TERMS AND PATTERNS

$$M,N,L ::= n \mid \mathsf{rd}\,(M) \mid \mathsf{wr}\,(M) \mid (M,\, N) \mid \mathsf{at}\,(M,\, N) \mid \{M : T\}$$
$$X ::= (x,\, y) \mid \mathsf{at}\,(x,\, y)$$

Terms include names as well as read and write capabilities, $\mathsf{rd}\,(M)$ and $\mathsf{wr}\,(M)$, which may be passed individually as in Odersky's polarized π [41]. The term $\mathsf{at}\,(M,\, N)$ is an attested message originating from hash M with payload N. The constructors for pairs and attestations each have a corresponding nonblocking destructor in the pattern language. The term $\{M : T\}$ carries a term M with asserted type T (cf. the dynamic types of [4]). As illustrated in section 2, terms of the form $\{M : T\}$ are used to convey type, and hence policy, information between processes that have no pre-established knowledge of one another's behavior or requirements, but the information can only be trusted when $\{M : T\}$ originates from a certified process. Attestation is used to ensure that such terms do originate from a certified process before secure channels are established.

PROCESSES AND CONFIGURATIONS

$$P,Q,R ::= \mathsf{iscert}\,M;\, P \mid \mathsf{typecase}\,\{x : T\} = M;\, P \mid \mathsf{let}\,X = M;\, P \mid \mathsf{scope}\,M\,\mathsf{is}\,\sigma$$
$$\mid M?x;\, P \mid M!N \mid \mathsf{new}\,a : T;\, P \mid P \mid Q \mid \mathsf{repeat}\,P \mid \mathsf{stop}$$
$$C,D,A,B ::= h\,[P] \mid \mathsf{new}_h\,a : T;\, C \mid C \mid D \mid \mathsf{stop}$$

The test $\mathsf{iscert}\,M$ succeeds if M is a certified hash, otherwise it blocks. The typecase $\mathsf{typecase}\,\{x : T\} = M$ succeeds if M is a term with an asserted type that is a subtype of T, otherwise it blocks. The expectation $\mathsf{scope}\,M\,\mathsf{is}\,\sigma$ asserts that the scope of M is limited by the hash formula σ; we discuss hash formulas with runtime errors below. The primitives for reading, writing, new names, concurrency, repetition and inactivity have the standard meanings from the asynchronous π calculus [11, 29]. The constructs for configurations are standard for located π-calculi [28]; note that the construct for new names records the identity of the process that created the name.

NOTATION. We identify syntax up to renaming of bound names. For any syntactic category with typical element e, we write $fn(e)$ for the set of free names occurring in e. We write $M\{^N/_x\}$ for the capture-avoiding substitution of N for x in M. For any syntactic category with typical element e, we write sequences as $\vec{e}$ and sets as $\bar{e}$. We occasionally extend this convention across binary constructs, for example writing $\vec{n}:\vec{T}$ for the sequence of bindings $n_1 : T_1, \ldots, n_m : T_m$. We sometimes write "_" for a syntactic element that is not of interest. □

The evaluation semantics is given in the chemical style [16] using a structural equivalence and small-step evaluation relation. (We write multistep evaluation as $\bar{g} \triangleright C \rightarrow^* D$.) We elide the definition of the structural equivalence, which is standard for located π-calculi [28]; for example, the rule for allowing new to escape from a process is "$h\,[\mathsf{new}\ a:T;\ P] \equiv \mathsf{new}_h\ a:T;\ h\,[P]$".

EVALUATION $(\bar{g} \triangleright C \rightarrow D)$

$$\frac{}{\bar{g} \triangleright f[\mathsf{wr}(a)\,!M] \mid h[\mathsf{rd}(a)?x;\ P] \rightarrow h[P\{^M/_x\}]} \qquad \frac{}{\bar{g} \triangleright h[\mathsf{iscert}\,f;\ P] \rightarrow h[P]}\ f \in \bar{g}$$

$$\frac{\text{if } h \in \bar{g} \text{ then } \bar{g}:\mathsf{Cert} \vdash T <: S}{\bar{g} \triangleright h[\mathsf{typecase}\ \{x:S\} = \{M:T\};\ P] \rightarrow h[P\{^M/_x\}]} \qquad \frac{X\{^{\vec{N}}/_{\vec{y}}\} = M}{\bar{g} \triangleright h[\mathsf{let}\ X{=}M;\ P] \rightarrow h[P\{^{\vec{N}}/_{\vec{y}}\}]}$$

$$\frac{\bar{g} \triangleright C \rightarrow D}{\bar{g} \triangleright C \mid B \rightarrow D \mid B} \qquad \frac{\bar{g} \triangleright C \rightarrow D}{\bar{g} \triangleright \mathsf{new}_h\ a;\ C \rightarrow \mathsf{new}_h\ a;\ D} \qquad \frac{\bar{g} \triangleright C \rightarrow D \quad C \equiv C'}{\bar{g} \triangleright C' \rightarrow D' \quad D \equiv D'}$$

The first rule allows communication between processes, in the standard way. The rule for iscert allows a process to verify that a hash is certified; in the residual, f is known to be a certified hash. The rule for typecase allows retrieval of data from a term with an asserted type. A dynamic subtype check enforces agreement between the asserted type T and the expected type S; subtyping is defined in section 4. The let rule is used to decompose attestations and pairs. The structural rules are standard.

Note that $\bar{g}$ is only required by typecase to allow opponents processes to avoid dynamic checks. The other uses of $\bar{g}$ (in iscert and typecase) can be removed, as is shown in the full version of the paper.

Runtime Error and Robust Safety. Our primary interest in typing is to enforce access control policies. Policies are specified in terms of hash formulas.

LATTICE OF HASH FORMULAS $(\rho \leq \sigma)$

$$\rho, \sigma ::= \mathsf{any} \mid \mathsf{cert} \mid h_1, \ldots, h_n \qquad \frac{}{\rho \leq \mathsf{any}} \quad \frac{}{\bar{h} \leq \mathsf{cert}} \quad \frac{}{\mathsf{cert} \leq \mathsf{cert}} \quad \frac{\bar{h} \subseteq \bar{g}}{\bar{h} \leq \bar{g}}$$

Hash formulas are interpreted using an open world assumption; we allow that not all programs nor typed programs are known. The special symbol cert is interpreted as a conservative approximation of the set of well-typed programs.

Access control policies are specified in scope expectations [21, 24]. We develop a notion of *runtime error* to capture access control and memory safety violations.

RUNTIME ERROR $(\bar{g} \triangleright C \xrightarrow{error})$

$$\frac{h \in \bar{g} \quad f \not\leq \sigma\{\bar{g}/\mathsf{cert}\}}{\bar{g} \triangleright h[\mathsf{scope}\, M \,\mathsf{is}\, \sigma] \mid f[M?x;\, P] \xrightarrow{error}} \quad \frac{h \in \bar{g} \quad M \neq \mathsf{rd}(_)}{\bar{g} \triangleright h[M?x;\, P] \xrightarrow{error}} \quad \frac{\bar{g} \triangleright C \xrightarrow{error}}{\bar{g} \triangleright C \mid D \xrightarrow{error}}$$

$$\frac{h \in \bar{g} \quad f \not\leq \sigma\{\bar{g}/\mathsf{cert}\}}{\bar{g} \triangleright h[\mathsf{scope}\, M \,\mathsf{is}\, \sigma] \mid f[M!N] \xrightarrow{error}} \quad \frac{h \in \bar{g} \quad M \neq \mathsf{wr}(_)}{\bar{g} \triangleright h[M!N] \xrightarrow{error}} \quad \frac{\bar{g} \triangleright C \xrightarrow{error}}{\bar{g} \triangleright \mathsf{new}_h\, a;\, C \xrightarrow{error}}$$

$$\frac{h \in \bar{g} \quad M \neq \{_:_\}}{\bar{g} \triangleright h[\mathsf{typecase}\, \{_:_\} = M;\, P] \xrightarrow{error}} \quad \frac{h \in \bar{g} \quad M \neq (_,_)}{\bar{g} \triangleright h[\mathsf{let}\, (_,_) = M;\, P] \xrightarrow{error}} \quad \frac{\bar{g} \triangleright C \xrightarrow{error}}{\bar{g} \triangleright C' \xrightarrow{error}}\; C \equiv C'$$

A runtime error occurs on certain shape errors and whenever a term is used outside of its allowed scope. For example, the following term is in error, since the certified process h is writing on a term of the wrong shape.

$$\{h,g\} \triangleright h[\mathsf{rd}(a)\,!n] \xrightarrow{error}$$

The following term is also in error, since the certified process h expects the scope of $\mathsf{wr}(a)$ to include only certified processes, yet the uncertified process f is attempting to write on a.

$$\{h,g\} \triangleright h[\mathsf{scope}\,\mathsf{wr}(a)\,\mathsf{is}\,\mathsf{cert}] \mid f[\mathsf{wr}(a)\,!n] \xrightarrow{error}$$

ROBUST SAFETY

A process is *h-initial* if every attested term has the form "$\mathsf{at}(h,\, M)$."
A configuration $h_1[P_1] \mid \cdots \mid h_\ell[P_\ell]$ is an *initial $\bar{g}$-attacker* if $\{h_1,\ldots,h_\ell\}$ is disjoint from $\bar{g}$ and every P_i is h_i-initial.
A configuration C is *$\bar{g}$-safe* if $\bar{g} \triangleright C \rightarrow^* D$ implies $\neg(\bar{g} \triangleright D \xrightarrow{error})$.
A configuration C is *robustly $\bar{g}$-safe* if $C \mid A$ is $\bar{g}$-safe for every initial $\bar{g}$-attacker A.

The statement of robust safety ensures that certified processes are error-free, even when combined with arbitrary attackers. The restriction that initial attacker $h[P]$ be h-initial requires only that attackers not forge attestations.

4 Statics

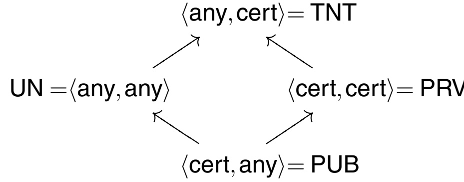
We describe a type system that ensures robust safety, i.e., runtime errors cannot occur even in the presence of attackers. Types also convey policy information—access-control policies specified in terms of hash formulas in this paper—that can be transmitted and tested at runtime. *Kinds* are assigned to types to restrict the use of unsafe types. In this section, we present parts of the type system and state the robust-safety theorem. The development is heavily influenced by Gordon and Jeffrey [23] and Haack and Jeffrey [25].

Policies, Types and Kinds. We assign to every channel type a policy regulating access to the channel. For channel types with policy $\langle\rho,\sigma\rangle$, ρ (respectively σ) controls the source (respectively destination) of the data communicated by the channel: thus ρ indicates the set of *writers*; σ indicates the set of *readers*. A *kind* is a policy $\langle\rho,\sigma\rangle$ in which ρ and σ are either cert or any.

POLICIES AND KINDS

$\rho,\ \sigma$ ::= $\mathsf{cert} \mid \mathsf{any} \mid \bar{h}$	Hash Formulas (Repeated)
$\alpha,\ \beta$ ::= $\mathsf{cert} \mid \mathsf{any}$	Kind Formulas
Φ, Ψ ::= $\langle \rho, \sigma \rangle$	Policies
$\mathcal{K}, \mathcal{J}$ ::= $\langle \alpha, \beta \rangle$	Kinds
$\mathsf{TNT} \triangleq \langle \mathsf{any}, \mathsf{cert} \rangle$	Tainted Secret Kind
$\mathsf{PRV} \triangleq \langle \mathsf{cert}, \mathsf{cert} \rangle$	Untainted Secret Kind
$\mathsf{UN} \triangleq \langle \mathsf{any}, \mathsf{any} \rangle$	Tainted Publishable Kind
$\mathsf{PUB} \triangleq \langle \mathsf{cert}, \mathsf{any} \rangle$	Untainted Publishable Kind
$\dfrac{\rho \le \rho' \text{ and } \sigma' \le \sigma}{\langle \rho, \sigma \rangle \le \langle \rho', \sigma' \rangle}$	Subpolicy Relation

The subpolicy relation, $\Phi \le \Psi$, indicates that Ψ is more restrictive than Φ. In more restrictive policies, it is always safe to overestimate the origin of a value and to underestimate its scope. That is, in $\langle \rho, \sigma \rangle$, ρ is an upper bound on origin, and σ is an lower bound on scope. When specialized to kinds, the subpolicy relation reduces to the subkinding relation from [25].



We write $\mathcal{K} \sqcup \mathcal{J}$ for the join operator over this lattice. For example $\mathsf{UN} \sqcup \mathsf{PRV} = \mathsf{TNT}$.

TYPES AND TYPING ENVIRONMENTS

T, S, U, R ::= $\mathsf{Hash} \mid \mathsf{Cert} \mid \mathsf{Top}\,\mathcal{K} \mid \mathsf{Dyn}\,(h)\,\mathcal{K} \mid (x : T,\ S)$
$\quad \mid \mathsf{Ch}\,\Phi\,(T) \mid \mathsf{Rd}\,\Phi\,(T) \mid \mathsf{Wr}\,\Phi\,(T)$

$\mathsf{Un} \triangleq \mathsf{Top}\,\mathsf{UN}$	Top of Kind UN
$\mathsf{Tnt} \triangleq \mathsf{Top}\,\mathsf{TNT}$	Top of Kind TNT
E ::= $n_1 : T_1, \ldots, n_m : T_m$	Typing Environments
$dom(\vec{n} : \vec{T}) \triangleq \vec{n}$	Domain of an Environment

Type Hash can be given to any hash, whereas Cert can be given only to certified hashes. $\mathsf{Top}\,\mathcal{K}$ is the type given to attestations containing data of kind $\mathcal{K}$. $\mathsf{Dyn}\,(h)\,\mathcal{K}$ is the type of dynamically-typed data that was attested by h. In the dependent pair type "$(x : T,\ S)$" x is bound with scope S; if $x \notin fn(S)$ we write simply $(T,\ S)$. The channel types indicate the policy Φ associated with the channel.

EXAMPLE. Although we allow creation of names at top types, these do not allow a full expression of access control policies. We provide an encoding of data, where $\mathsf{Data}\,(\sigma)$ is the type of data visible to σ.

$$\mathsf{Data}\,(\sigma) \triangleq \mathsf{Wr}\langle\sigma,\mathsf{any}\rangle\,(\mathsf{Un}) \qquad\qquad \mathsf{display}\, M \triangleq \mathsf{new}\; n:\mathsf{Un};\; M\,!\,n$$

A simple use of data is: $\mathsf{new}\; n:\mathsf{Data}\,(\mathsf{cert})\,;\; \mathsf{display}\; n$. □

Judgments. The following judgments are used in the typing system. Due to space limitations, we discuss only the most important rules.

JUDGMENTS

$E \vdash \diamond$	Well-Formed Typing Environments
$E \vdash \Phi :: \mathcal{K}$	Well-Formed Policies
$E \vdash T :: \mathcal{K}$	Well-Formed Types of Kind $\mathcal{K}$
$E \vdash T <: S$	Subtyping
$E \vdash M : T$	Well-Formed Terms of Type T
$E \vdash_h P$	Well-Formed Process at h

The rules for environments require that every type in the environment be well-formed. A policy is well-formed with respect to a typing environment if every hash in the policy has type Cert in the environment.

$$\frac{E \vdash \Phi :: \langle \mathsf{cert},\alpha\rangle \quad E \vdash T :: \langle _,\beta\rangle \quad \alpha \le \beta}{E \vdash \mathsf{Rd}\,\Phi\,(T) :: \langle \mathsf{cert},\alpha\rangle} \qquad \frac{E \vdash \Phi :: \langle \mathsf{any},\alpha\rangle \quad T = \mathsf{Top}\langle \mathsf{any},\beta\rangle \quad \alpha \le \beta}{E \vdash \mathsf{Rd}\,\Phi\,(T) :: \langle \mathsf{any},\alpha\rangle}$$

The rules for well-formed types require that read capabilities of kind UN receive values (at a type of) of kind UN; those of kind PUB receive values of kind UN or PUB; those of kind TNT receive values of kind UN or TNT; and those of kind PRV receive values of any kind. Write capabilities are similar for UN and PRV, but differ at the other kinds. Write capabilities of kind PUB send values of kind UN or TNT; those of kind TNT send values of kind UN or PUB. In the analogous rules for write capabilities, the kind is inverted with respect to the policy. As a consequence, if a channel communicates untainted data then the write capability is given at most trusted scope; if a write capability is publishable, then the data it communicates is tainted.

Subtyping. Subtyping is reflexive and transitive, with top types at each kind. Read and write capability types must have related policies in order to be related by subtyping.

$$\frac{\Phi \le \Psi \quad E \vdash T <: S \quad E \vdash S :: \mathit{kind}(\Psi)}{E \vdash \mathsf{Rd}\,\Phi\,(T) <: \mathsf{Rd}\,\Psi\,(S)} \qquad \frac{\Psi \le \Phi \quad E \vdash S <: T \quad E \vdash S :: \mathit{kind}(\Psi)}{E \vdash \mathsf{Wr}\,\Phi\,(T) <: \mathsf{Wr}\,\Psi\,(S)}$$

In the read rule, the requirement that S be well-formed is necessary since Ψ may be tainted even if Φ is not. Likewise in the write rule, Ψ may be publishable even if Φ is not.

Typing and Robust Safety. The typing rules are designed to ensure robust safety whilst allowing typechecked processes to have limited interaction with processes that are not known to be typechecked. The interesting rules for terms are those for dynamic types and attestations.

$$\frac{E \vdash M : T \quad E \vdash h : \mathsf{Cert} \quad E \vdash T :: \mathcal{K}}{E \vdash \{M : T\} : \mathsf{Dyn}\,(h)\;\mathcal{K}} \qquad \frac{E \vdash M : \mathsf{Cert} \quad E \vdash N : \mathsf{Dyn}\,(M)\;\mathcal{K}}{E \vdash \mathsf{at}\,(M, N) : \mathsf{Top}\;\mathcal{K}}$$

The rule for dynamic types constrains each type assertion to be associated with the hash of a typechecked process, and that hash is recorded in the (dependent) dynamic type. The rule for attestations constrains the hash in an attestation to be the same as the one used in the inner type assertion.

Turning to processes, consider the following three rules.

$$\frac{\begin{array}{l} E \vdash M : \mathsf{Top}\;\mathcal{K} \\ E, x : \mathsf{Hash}, y : \mathsf{Dyn}\,(x)\;\mathcal{K} \models_{\bar{h}} P \end{array}}{E \models_{\bar{h}} \mathsf{let\ at}\,(x, y) = M;\ P} \quad \frac{\begin{array}{l} T \in \{\mathsf{Hash}, \mathsf{Cert}\} \\ E, x : \mathsf{Cert}, E' \models_{\bar{h}} P \end{array}}{E, x : T, E' \models_{\bar{h}} \mathsf{iscert}\ x;\ P} \quad \frac{\begin{array}{l} E \vdash M : \mathsf{Dyn}\,(f)\;\mathcal{K} \\ E \vdash f : \mathsf{Cert} \\ E, x : S \models_{\bar{h}} P \end{array}}{E \models_{\bar{h}} \mathsf{typecase}\ \{x : S\} = M;\ P}$$

The pattern-matching rule for attestations is used to decompose an attestation into a hash and a dynamic type associated with that hash. However, the term of dynamic type cannot be unpacked (using the rule for typecase) until the hash is known to correspond to a well-typed process. This is established using iscert, leading to the pattern seen in the example of section 2 of an attestation decomposition, followed by a hash check, and finally a typecase.

$$\frac{E \vdash \diamond \quad E \vdash M : \mathsf{Ch}\,\langle \rho, \sigma \rangle\,(T)}{E \models_{\bar{h}} \mathsf{scope\ rd}\,(M)\ \mathsf{is}\ \sigma} \qquad \frac{E \vdash M : \mathsf{Rd}\,\langle _, \sigma \rangle\,(T) \quad E, x : T \models_{\bar{h}} P \quad h \leq \sigma}{E \models_{\bar{h}} M?x;\ P}$$

The type rule for read scoping expectations forces the process making the expectation to know the channel type, and not just the read capability type. This is necessary because policies are invariant on channel types but covariant on read channel types, so a process that only knows the read capability type may have a poor approximation of the actual policy that is used elsewhere in a configuration. To avoid access control violations, the rule for reading processes requires that the process has authorization to read from a read capability. Although a static authorization check may initially appear restrictive, note that the static authorization check may follow a dynamic subtyping check for a read capability received from another process.

The robust safety theorem states that processes can safely be typechecked and deployed independently without any shared untainted or secret data (such as public or secret keys), even in the presence of attackers.

THEOREM (ROBUST SAFETY). *Let E be an environment in which every type is generative and can be given kind* UN. *Let g_i and P_i be defined such that ($1 \leq i \leq n$):*

P_i is g_i-initial and $E, \bar{h} : \mathsf{Cert}, \bar{f} : \mathsf{Hash} \models_{g_i} P_i$ for some $\bar{h} \subseteq \{g_1, \ldots, g_n\}$ and $\bar{f}$.

Then $g_1\,[P_1] \mid \cdots \mid g_n\,[P_n]$ is robustly $\{g_1, \ldots, g_n\}$-safe.

Proof sketch. The proof requires an invariant that is implied by initial typing and preserved by reduction. We formalize the invariant as a more liberal typing system recording the sets of certified and opponent hashes. The central lemmas are *Certified typability:* All certified processes are well typed. *Opponent typability:* All opponent processes are well typed. *Preservation:* Well typing is preserved by evaluation. *Progress:* Well typed terms cannot give rise to runtime errors. □

5 Related Work

Remote Attestation. NGSCB and the TCG have provoked considerable controversy. For example, see [12, 14].

Abadi [2] outlines a broad range of trusted hardware applications that use remote attestation to convey trust assertions from one process to another. Our work can be seen as a detailed formal study of a specific kind of trust assertion, namely information about the type and access control policy for communicated data.

The NGSCB [37, 38, 39] remote attestation mechanism, and the TCG [50, 42, 15] hardware that underpins it, are more complex than the π-rat remote attestation mechanism. We have omitted much of the complexity in order to focus on the core policy issues. For a logical description of NGSCB's mechanism see [10]. For a concrete account of implementing NGSCB-like remote attestation on top of TCG hardware see [45].

Haldar, Chandra, and Franz [26, 27] use a virtual machine to build a more flexible remote attestation mechanism on top of the primitive remote attestation mechanism that uses hashes of executables. In their system, a process requesting an attestation from a second process can send test code to execute on the second process's virtual machine and ask for the results to be reported in attestations. Sadeghi and Stüble [44] observe that systems using remote attestation may be fragile, and discuss a range of options for implementing more flexible remote attestation mechanisms based upon system properties (left unspecified as the focus is upon implementation strategies). Sandhu and Zhang [46] consider the use of remote attestation to protect disseminated information.

Process Calculi. As discussed in the introduction, π-rat builds upon existing work [9, 3, 22, 23, 25] with symmetric-key and asymmetric-key cryptographic primitives in pi-calculi. Notably, the kinding system is heavily influenced by the pattern-matching spi-calculus [25]. Our setting is quite different, however. In particular, processes establish their own secure channels and corresponding policies, as opposed to relying upon a mutually-trusted authority to distribute initial keys and policies. In addition, the access control policies used here are not immediately expressible in spi, since processes do not have associated identity. The techniques used to verify authenticity and other properties as in [22, 21] should be applicable to π-rat, though we make no attempt to address authenticity or replay attacks here. Finally, our primitive for checking attestation includes an implicit notion of *authorization* which is made explicit in [25]. Scaling up to explicit authorizations would allow the possibility of enforcing policies that require multiple authorizations for certain actions.

There is some similarity between our work and that on the distributed π calculus [28]. In Dπ, locations are primarily collections of resources. Here, instead, we view "locations" as principals whose identity is determined by the actual code running. This is a different view of locality, determined less by where the code happens to be running and more on the identity of the code itself.

Because of the close relation between process terms and their hashes, attestation does not appear to fit neatly into existing abstract frameworks for π-calculi, such as applied π [5].

Code Identity. Code identity is also used in stack inspection [51] and other history-based access control policies [6]. Remote attestation can be used to implement similar policies in a distributed environment, but we leave this for future work.

Separate Compilation and Typechecking. The π-rat type system allows executables to be typechecked independently and subsequently linked together. Separate compilation and linkability is not a new idea in programming languages, see, e.g., [20], but is uncommon in spi-like calculi because there is usually a need to reliably distribute some shared secret or untainted data between separate processes in accordance with a type (policy). Recently Bugliesi, Focardi, and Maffei [18, 19] have considered separate typechecking in the context of a spi-like calculus.

Trusted Hardware. We have assumed that trusted hardware is trustworthy. Amongst other things, the trusted hardware must correctly protect the memory of processes from attackers, attackers must not be able to access the trusted hardware key, and processor manufacturers must not issue fake certificates and keypairs to anyone (such as law enforcement, intelligence agencies, or data recovery firms). For accounts of the difficulties involved in creating such trusted hardware see [13, 31] for an attacker's perspective and [15, 48] for a defender's perspective. Irvine and Levin [32] provide a warning about placing too much trust in the integrity of COTS.

Other research efforts on implementations of trusted hardware, such as [35, 36, 47], are orthogonal to the work presented here.

6 Conclusion

This paper is an early contribution to the study of remote attestation in programming languages. We have defined an extension of the π-calculus with a remote attestation primitive and access control assertions for channels. Executables may be typechecked and deployed individually, which is a significant advantage for the intended applications of trusted hardware. The resulting typechecked configurations discover and obey access control policies even with the addition of opponents. To the best of our knowledge, this is the first paper to provide static analysis principles for building systems that use the remote attestation mechanism.

By incorporating higher-order communication, one could reason about runtime certification of executables and the distribution of knowledge of the certification. The presence of hashes identifying processes also makes it possible to imagine recovering traditional memory safety without sacrificing opponent typability.

It would be useful to extend π-rat with access control policies using linked namespaces that denote sets of trusted hashes as opposed to sets of public keys in the RT framework [34]. With development tools that also run on trusted hardware, there are some interesting new possibilities. For example, we might use a compiler (not necessarily modeled as a well-typed π-rat executable) that issues an attestation associating the hash of the source code and the hash of the resulting executable. An access control policy might state that a well-typed executable must have been derived from source code signed by a trusted developer's private key, where that developer is expected to follow certain procedures to provide a degree of assurance. The use of development tools that attest to their output would help to mitigate the threat of Trojan horses in tools, see [49].

Acknowledgment. We would like to thank Christian Haack, Radha Jagadeesan, Alan Jeffrey, Will Marrero, William Pollock, Daniel Sweeney, and the anonymous referees for their comments.

References

1. M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5), 1999.
2. M. Abadi. Trusted computing, trusted third parties, and verified communications. In *SEC2004: 19th IFIP International Information Security Conference*, 2004.
3. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3), 2003.
4. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2), 1991.
5. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL '01*, 2001.
6. M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.
7. M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *POPL '00*, 2000.
8. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Inf. Comput.*, 174(1), 2002.
9. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1), 1999.
10. M. Abadi and T. Wobber. A logical account of NGSCB. In *FORTE '04*, 2004.
11. R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations of the asynchronous π-calculus. *Theor. Comput. Sci.*, 195(2), 1998.
12. R. Anderson. 'Trusted Computing' Frequently Asked Questions. http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html, 2003. Version 1.1.
13. R. Anderson and M. Kuhn. Tamper resistance - a cautionary note. In *Second USENIX Workshop on Electronic Commerce Proceedings*, 1996.
14. W. A. Arbaugh. Improving the TCPA specification. *IEEE Computer*, 2002.
15. W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, 1997.
16. G. Berry and G. Boudol. The chemical abstract machine. In *POPL '90*, 1990.
17. M. Bugliesi, S. Crafa, A. Prelic, and V. Sassone. Secrecy in untrusted networks. In *ICALP '03*, 2003.
18. M. Bugliesi, R. Focardi, and M. Maffei. Compositional analysis of authentication protocols. In *ESOP*, 2004.
19. M. Bugliesi, R. Focardi, and M. Maffei. Analysis of typed analyses of authentication protocols. In *CSFW*, 2005.
20. L. Cardelli. Program fragments, linking, and modularization. In *POPL '97*, 1997.
21. C. Fournet, A. Gordon, and S. Maffeis. A type discipline for authorization policies. In *ESOP '05*, 2005.
22. A. D. Gordon and A. S. A. Jeffrey. Authenticity by typing for security protocols. *J. Computer Security*, 11(4), 2003.
23. A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *J. Computer Security*, 12(3/4), 2004.
24. A. D. Gordon and A. S. A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR*, 2005.

25. C. Haack and A. S. A. Jeffrey. Pattern-matching spi-calculus. In *Proc. IFIP WG 1.7 Workshop on Formal Aspects in Security and Trust*, 2004.
26. V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX VM*, 2004.
27. V. Haldar and M. Franz. Symmetric behavior-based trust: A new paradigm for internet computing. In *New Security Paradigms Workshop*, 2004.
28. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173, 2002.
29. K. Honda and M. Tokoro. On asynchronous communication semantics. In *ECOOP '91: Proceedings of the Workshop on Object-Based Concurrent Computing*, 1992.
30. K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP '00*, 2000.
31. A. Huang. *Hacking the Xbox*. Xenatera Press, 2003.
32. C. Irvine and T. Levin. A cautionary note regarding the data integrity capacity of certain secure systems. In *Integrity, Internal Control and Security in Information Systems*, 2002.
33. X. Leroy and M. Mauny. Dynamics in ML. In *FPCA*, 1991.
34. N. Li and J. Mitchell. RT: A role-based trust-management framework. In *DARPA Information Survivability Conference and Exposition (DISCEX III)*, 2003.
35. D. Lie, C. Thekkath, P. Lincoln, M. Mitchell, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS-IX*, 2000.
36. D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *19th ACM Symposium on Operating Systems Principles*, 2003.
37. Microsoft. Longhorn developer preview documentation. Distributed at Microsoft's Professional Developers Conference in Los Angeles, 2003.
38. Microsoft. NGSCB: TCB and software authentication, 2003.
39. Microsoft. Security model for the Next-Generation Secure Computing Base, 2003.
40. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1), 1992.
41. M. Odersky. Polarized name passing. In *FST-TCS '95*, 1995.
42. S. Pearson, editor. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, 2002.
43. J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. *J. Automated Reasoning*, 31(3–4), 2003.
44. A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *New Security Paradigms Workshop*, 2004.
45. R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *13th USENIX Security Symposium*, 2004.
46. R. Sandhu and X. Zhang. Peer-to-peer access control architecture using trusted computing technology. In *SACMAT*, 2005.
47. A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.
48. S. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31, 1999. Special Issue on Computer Network Security.
49. K. Thompson. Reflections on trusting trust. *CACM*, 27(8), 1984.
50. Trusted Computing Group. Trusted Computing Platform Alliance: Main specification, version 1.1b. http://www.trustedcomputinggroup.org, 2003.
51. D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4), 2000.

Distributed Unfolding of Petri Nets*

Paolo Baldan[1], Stefan Haar[2], and Barbara König[3]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] INRIA Rennes, *Distribcom* team, France
[3] Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany

Abstract. Some recent Petri net-based approaches to fault diagnosis of distributed systems suggest to factor the problem into local diagnoses based on the unfoldings of local views of the system, which are then correlated with diagnoses from neighbouring supervisors. In this paper we propose a notion of system factorisation expressed in terms of pullback decomposition. To ensure coherence of the local views and completeness of the diagnosis, data exchange among the unfolders needs to be specified with care. We introduce interleaving structures as a format for data exchange between unfolders and we propose a distributed algorithm for computing local views of the unfolding for each system component. The theory of interleaving structures is developed to prove correctness of the distributed unfolding algorithm.

1 Introduction

Partial order semantics are often instrumental in providing a compact representation of the behaviour of concurrent systems: modelling concurrency of events in an explicit way rather than considering all the possible interleavings of such events helps in tackling the so-called state explosion problem. In recent years there has been a growing interest in the use of unfolding-based approaches. Originally introduced in the setting of Petri nets, the unfolding semantics [14] is a branching partial order semantics which represents in a single structure all the possible events in computations, and their causal dependencies and conflicts (branching points). Each branch represents a concurrent execution of the net, in the form of an acyclic conflict-free net. The unfolding provides an "efficient" representation of the state space of the system, not only taking advantage of a partial order representation of concurrency, but also keeping together different computations in its branching structure until a conflict is reached.

When analysing a complex system it happens frequently that we want to consider only a small part of such a system: either because the system is inherently distributed and each observer has access only to a local component of it, or because the system is too big to be analysed or monitored as a whole. In this paper, we take Petri nets as systems models and their unfolding semantics as reference

* Partially supported by EC RTN 2-2001-00346 SegraVis, MIUR project PRIN 2005015824 ART, European NoE ARTIST (IST-2001-34820), French RNRT project SWAN (No. 03 S 481), DFG project SANDS, SFB 627 (NEXUS).

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 126–141, 2006.
© Springer-Verlag Berlin Heidelberg 2006

semantics, and we view systems as made up of smaller components. Then we show how the projection of the semantics of the whole system over each single local component can be computed locally via a distributed unfolding algorithm, requiring only a minimal interaction among components.

The original motivation of this work is not verification, but distributed diagnosis of asynchronous systems. The general principle of diagnosis for discrete event systems (DES) can be stated as follows: not all transitions of a system are observable; in particular, faults are invisible and have to be deduced from the observations. This deduction of behaviour from the observations is the topic of diagnosis; efforts to force the system into a desired behaviour are studied in the domain of *control*. Although it has an important intersection with diagnosis, control is a clearly distinct problem, not addressed by the present article. Diagnosis can be approached via the construction of finite automata, the *diagnosers*, detailed in [19]; the input of a diagnosis procedure is the language of observed sequences and its output is the language of behaviours that explain the observations. Communication among diagnosers allows for a decentralised diagnosis, in which different diagnoses are proposed by various local diagnosers and then merged to filter out incompatible local views (see, e.g., [6, 16, 5]). Some authors consider timed extensions [17] or use Petri nets as system models [10].

The diagnosis approach in [3, 4, 9], which the present work builds upon, differs from all of the above in the fact that the asynchronous behaviour is captured by partial order semantics, thus abstracting away time aspects and interleavings of concurrent events in order to fight state space explosion. The system behaviour is given in the form of a Petri net model, where only a subset of transitions is observable. Then a sequence (or partially ordered scenario) of observations, called *alarm pattern*, is explainable by several net computations. These explanations are obtained by unfolding the synchronous product of the model net with the alarm pattern, and extracting the maximal configurations compatible with the alarm pattern (see [3]). This approach suffers, for large systems, from the explosion of the size of the global unfolding. Moreover, the practice in diagnosis for large networks justifies the use of several supervisors having only a partial view of the network.

This leads to the idea of *distributed diagnosis via unfoldings*: each supervisor computes a local diagnosis and an exchange of messages with neighbouring supervisors allows to eliminate branches that do not appear as local traces of admissible global configurations. Being able to construct the local views of the global unfolding (of prohibitive size in general) without computing it is the heart of the problem. We remark that we are interested in the projections over the local components of the unfolding of the whole system rather than in the unfoldings of the components themselves. This will become clearer in the technical treatment, but intuitively the reason is that the "autonomous" unfolding of each single component would include "spurious" runs which, although consistent with the structure of the local component itself, have no counterpart in the behaviour of the whole system, due to component interactions. In [4, 9] Petri net components were fused on places, and the fusion of views was done through products

of event structures obtained by using a projection operation with an exchange of messages relating transition actions. Another fusion approach using so-called *augmented processes* is developed in [7, 8].

At a technical level, we will introduce a decomposition/composition mechanism based on pullbacks which allows to view a given Petri net N_3 as built as the join of two components N_1 and N_2 (or more) along a common interface net N_0. The categorical approach allows to exploit a compositionality result which plays a basic role in the design of the distributed unfolding algorithm: the unfolding construction can be expressed as a right adjoint functor between suitable categories of nets and thus it preserves pullbacks. Hence the unfolding of a net N_3, arising as the the pullback of N_1 and N_2 along N_0, can be obtained as the pullback of the unfoldings of the single components.

In order to compute the projections of the full unfolding over the various net components, we propose a distributed algorithm requiring an exchange of information among such components. The components communicate through the interface net, whose unfolding is used to store information about dependencies on events induced by both components. This information is conceptually stored in so-called interleaving structures, whose theory provides a solid theoretical basis for proving the correctness of the distributed unfolding procedure. More specifically, factorisation results from category theory will be used to show that the information stored in the interface suffices in order to obtain the desired result.

The paper is organised as follows. In §2 we lay some general technical ground for the categorical techniques involved. In §3 we focus on Petri net decomposition, while in §4 we introduce Petri net unfoldings. In §5 we develop the theory of interleaving structures, which play a basic role in the distributed unfolding algorithm presented in §6. Finally, in §7 we draw some conclusions.

2 Notation and Categorical Background

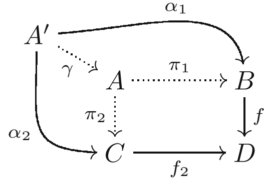
Given a (possibly partial) function $f : A \dashrightarrow B$ and $a \in A$ we will write $f(a)\downarrow$ whenever f is defined on a and $f(a)\uparrow$, otherwise.

Let A be a set. The powerset of A is denoted by $\mathbf{2}^A$. A *multiset* of A is a total function $M : A \to \mathbb{N}$. It is called *finite* if the underlying set $\{a \in A \mid M(a) > 0\}$ is finite. A finite multiset is sometimes denoted as a formal sum $M = \bigoplus_{a \in A} M(a) \cdot a$. The set of finite multisets of A is denoted by μA. A subset $X \subseteq A$ will be often treated as the multiset $\bigoplus_{a \in X} 1 \cdot a$.

A *(finitary) multirelation* $f : A \leftrightarrow B$ is a multiset of $A \times B$ such that for all $a \in A$ the set $\{b \in B \mid f(a, b) > 0\}$ is finite. Any multirelation $f : A \leftrightarrow B$ induces a function $\mu f : \mu A \to \mu B$ defined by $\mu f(\bigoplus_{a \in A} n_a \cdot a) = \bigoplus_{b \in B}(\sum_{a \in A} n_a\, f(a, b)) \cdot b$. We say that a multirelation $f : A \leftrightarrow B$ is *total* if for any $a \in A$ there exists $b \in B$ such that $f(a, b) > 0$, *injective* if for any $b \in B$ we have $\sum_{a \in A} f(a, b) \leq 1$, *surjective* if for any $b \in B$ we have $\sum_{a \in A} f(a, b) \geq 1$.

We will refer to some categorical concepts (see also [1]), and in particular we will make extensive use of pullbacks and factorisation structures.

Definition 1 (pullback). *Let* $\mathbf{C}$ *be a category and* $f_1 : B \to D$, $f_2 : C \to D$ *be arrows in* $\mathbf{C}$. *The* pullback *of* f_1 *and* f_2 *is an object* A *(*pullback object*) and a pair of arrows* $\pi_1 : A \to B$, $\pi_2 : A \to C$ *such that (i)* $f_1 \circ \pi_1 = f_2 \circ \pi_2$ *and (ii) for any object* A' *with arrows* $\alpha_1 : A' \to B$, $\alpha_2 : A' \to C$ *such that* $f_1 \circ \alpha_1 = f_2 \circ \alpha_2$ *there exists a unique arrow* $\gamma : A' \to A$ *such that* $\pi_i \circ \gamma = \alpha_i$ *(*$i \in \{1, 2\}$*).*



For instance, for a fixed a set Λ of labels, consider the category $\mathbf{LSet}^*$ of Λ-labelled sets and partial functions. Objects are pairs (A, λ), where A is a set and $\lambda \colon A \to \Lambda$ is a total labelling function, while arrows are label-preserving partial functions. Given two arrows $f_1 \colon (B, \lambda_B) \to (D, \lambda_D)$, $f_2 \colon (C, \lambda_C) \to (D, \lambda_D)$ the pullback object is (A, λ_A) with

$$
\begin{aligned}
A = {} & \{(b, c) \mid b \in B, c \in C, f_1(b) = f_2(c)\} \\
& \cup \{(b, *) \mid b \in B, f_1(b) \uparrow\} \cup \{(*, c) \mid c \in C, f_2(c) \uparrow\} \\
& \cup \{(b, c) \mid b \in B, c \in C, f_1(b), f_2(c) \uparrow \text{ and } \lambda_B(b) = \lambda_C(c)\}
\end{aligned}
$$

and λ_A, π_1 and π_2 defined in the obvious way.

Definition 2 (factorisation structures). *Let* $\mathbf{C}$ *be a category and let* E, M *be classes of morphisms in* $\mathbf{C}$*, closed under composition with isomorphisms. The pair* (E, M) *is called a* factorisation structure *for morphisms in* $\mathbf{C}$ *and* $\mathbf{C}$ *is called* (E, M)-structured *whenever*

- $\mathbf{C}$ *has* (E, M)-factorisations *of morphisms, i.e., each morphism* f *of* $\mathbf{C}$ *has a factorisation* $f = m \circ e$ *with* $e \in E$ *and* $m \in M$.
- $\mathbf{C}$ *has the unique* (E, M)-diagonalisation property*, i.e., for each commutative square as shown on the left-hand side below with* $e \in E$ *and* $m \in M$ *there exists a unique* diagonal*, i.e., a morphism* d *such that the diagram on the right-hand side commutes (i.e., such that* $d \circ e = f$ *and* $m \circ d = g$*).*

The classical example of (E, M)-factorisation in **Set** is the factorisation of a function f into a surjective and an injective part. In the following, morphisms from E are drawn using double-headed arrows of the form $A \twoheadrightarrow B$, whereas morphisms from M are drawn using arrows of the form $A \rightarrowtail B$.

In any (E, M)-structured category (E, M)-factorisations of morphisms are unique up to isomorphism, the sets E and M are both closed under composition and all arrows in M are stable under pullback.

3 Composing Petri Nets

In this section we introduce the basics of Petri nets and the corresponding category. Then we present a technique for decomposing Petri nets into smaller components (or equivalently to compose Petri nets) along a given interface, showing how the operation can be interpreted, in categorical terms, as a pullback.

We will consider labelled Petri nets, with morphisms as introduced in [20]. In the rest of the paper Λ denotes a fixed label set for all considered Petri nets.

Definition 3 (Petri net). *A* Petri net *is a tuple* $N = (S, T, \lambda, {}^\bullet(), ()^\bullet, m)$ *where* S *is the set of places,* T *is the set of transitions,* $\lambda: T \to \Lambda$ *is a labelling function,* ${}^\bullet(), ()^\bullet: T \to \mathbf{2}^S$ *associate to each transition* $t \in T$ *its pre-set and post-set, respectively, and* $m \in \mathbf{2}^S$ *is the initial marking.*

A (Petri net) morphism $\tau = (\eta, \beta): N \to N'$ *is a pair consisting of a partial function* $\eta: T \dashrightarrow T'$ *and a finitary multirelation* $\beta: S \leftrightarrow S'$ *such that (i)* $\mu\beta(m) = m'$*, and (ii) for any* $t \in T$*,* $\mu\beta({}^\bullet t) = {}^\bullet\eta(t)$ *and* $\mu\beta(t^\bullet) = \eta(t)^\bullet$*, where conventionally* ${}^\bullet\eta(t) = \eta(t)^\bullet = \emptyset$ *when* $\eta(t)\uparrow$*. The category of Petri nets and their morphisms is denoted by* **PN**.

In the sequel we will assume that in any considered Petri net, all transitions have a non-empty pre-set, a typical property required in unfolding-based approaches. Moreover we will denote the components of a Petri net N as S, T, λ, ${}^\bullet()$, $()^\bullet$ and m. Superscripts will carry over to the component names.

Example: Examples of Petri nets can be found in Fig. 1. Initially marked places are drawn with thick lines. Both nets consist of a loop involving four transitions, labelled over the set $\Lambda = \{\alpha, \beta, \gamma, \delta\}$.

For defining formally the local projections of the full unfolding we need some special classes of morphisms.

Definition 4 (projection and embedding). *A Petri net morphism* $\tau = (\eta, \beta) : N \to N'$ *is called a* projection *whenever* η *and* β *are surjective. It is called an* embedding *if both* η *and* β *are total and injective.*

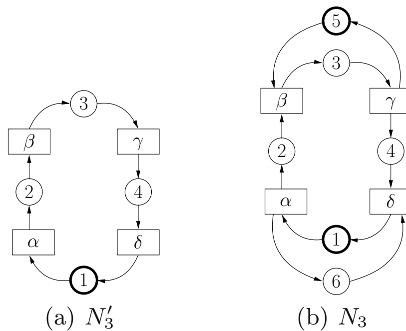


Fig. 1. Two examples of Petri nets

It can be shown that **PN** is (projection,embedding)-structured (*pe*-structured, for short). Given a **PN** morphism $\tau = (\eta, \beta) : N \to N'$, let $\tau(N)$ denote the subnet of N' including only transitions in $\eta(T)$. Then the projection $\tau : N \to \tau(N)$ and the inclusion of $\tau(N)$ into N' provide a *pe*-factorisation of τ.

In the following we define how to restrict a Petri net to a subset S_0 of its places. Specifically, a transition t will appear in the new net only if it is connected to at least one place in S_0.

Definition 5 (restricting a net). *Let N be a net and let $S_0 \subseteq S$ be a subset of places. Then the* restriction *of N to S_0, denoted $[N]_{S_0} = (S_0, T_0, \lambda_0, {}^\bullet(), ()^\bullet, m_0)$, is defined as follows: $T_0 = \{(t, 0) \mid t \in T \wedge ({}^\bullet t \cap S_0 \neq \emptyset \;\vee\; t^\bullet \cap S_0 \neq \emptyset)\}$, $\lambda_0((t, 0)) = \lambda(t)$, ${}^\bullet(t, 0) = {}^\bullet t \cap S_0$, $(t, 0)^\bullet = t^\bullet \cap S_0$ and $m_0 = m \cap S_0$.*

This induces a morphism $\tau_{N,S} = (\eta, \beta)\colon N \to [N]_S$ with $\eta(t) = (t, 0)$, whenever $(t, 0) \in T_0$, and $\eta(t) \uparrow$ otherwise. Furthermore $\beta(s, s') = 1$, whenever $s = s' \in S_0$, and $\beta(s, s') = 0$ otherwise.

The next proposition provides a recipe for decomposing Petri nets along some chosen places, which play the role of interface between the subcomponents.

Proposition 6 (decomposition of Petri nets). *Let N_3 be a Petri net and let $S_3 = S_1 \cup S_2$. Let $S_0 = S_1 \cap S_2$ and construct nets N_0, N_1, N_2 as follows: $N_1 = [N_3]_{S_1}$, $N_2 = [N_3]_{S_2}$, $N_0 = [N_1]_{S_0} = [N_2]_{S_0}$. Say that transitions in $T_i - T_0$ are* local *to N_i for $i \in \{1, 2\}$ and assume that transitions local to different nets have distinct labels. Then the nets N_i with $i \in \{0, 1, 2, 3\}$ together with the morphisms τ_{N_3,S_1}, τ_{N_3,S_2}, τ_{N_1,S_0}, τ_{N_2,S_0} form a pullback diagram.*

Note that, in order to exploit the results about unfolding, also the component nets must not contain transitions with empty pre-sets. Henceforth, all decompositions are supposed to have this property. For safe nets this can be achieved by introducing extra complement places. Also, decomposition will have to be done in such a way that local transitions in different components have different labels.

Example: Consider the Petri net N_3' in Fig. 1(a). We intend to split the loop along the places 1 and 3, i.e., we plan to decompose as described in Proposition 6 with $S_1 = \{1, 2, 3\}$ and $S_2 = \{1, 3, 4\}$. However, this would result in subcomponents N_0, N_1 and N_2 including transitions with empty pre-set. In order to avoid this problem, we can complement the interface places 1, 3 by adding two more places 5, 6, thus obtaining the net N_3 in Fig. 1(b). Call place $\bar{p}$ the *complement* of place p if $p^\bullet = {}^\bullet\bar{p}$, $\bar{p}^\bullet = {}^\bullet p$, and $p \in m \Leftrightarrow \bar{p} \notin m$. Then 5, 6 are complements for 1, 3. The new net is equivalent to N_3' (in a sense which can be formalised [15]) and can be safely decomposed using $S_1 = \{1, 2, 3, 5, 6\}$ and $S_2 = \{1, 3, 4, 5, 6\}$.

We split N_3 into two subnets N_1, N_2 with interface N_0 (according to Proposition 6), thus obtaining the pullback in category **PN** shown in Fig. 2.

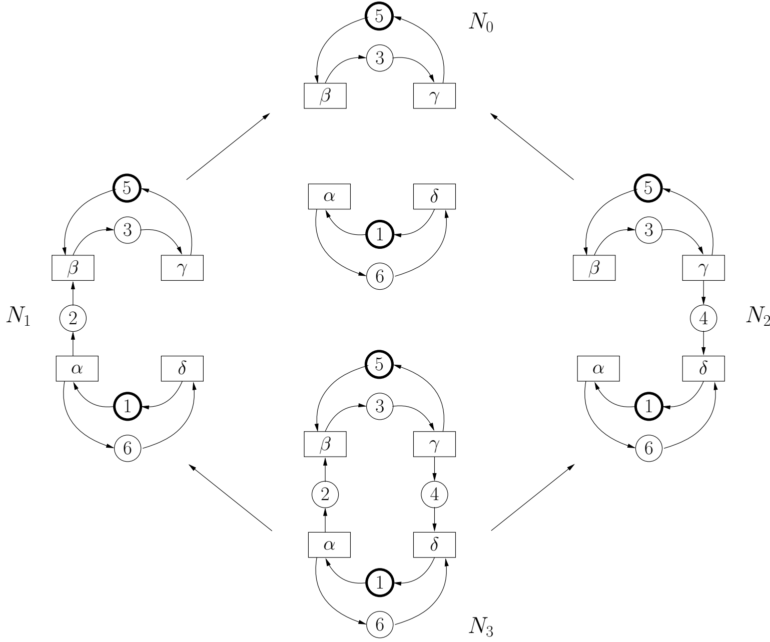


Fig. 2. Decomposing a loop as a pullback of nets

4 Unfolding Petri Nets

Recall that given a Petri net N the dependencies between transitions are captured by two basic relations, causality and conflict. *Causality* is the least transitive relation $<_N$ over $S \cup T$ such that if $s \in {}^\bullet t$ then $s <_N t$ and if $s \in t^\bullet$ then $t <_N s$. We denote by $\leq_N$ the reflexive closure of $<_N$ and for any $x \in S \cup T$, $\lfloor x \rfloor = \{y \in S \cup T : y \leq_N x\}$. *Conflict* is the least symmetric relation $\#_N$ over $S \cup T$ such that (i) if ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and $t \neq t'$ then $t \#_N t'$ and (ii) if $t \#_N t'$ and $t <_N t''$ then $t'' \#_N t'$.

Occurrence nets are basically acyclic nets where each place is generated by at most one transition. They are used to unfold Petri nets as described below.

Definition 7 (occurrence net). *An* occurrence net *is a net N satisfying:*

1. *if $t^\bullet \cap t'^\bullet \neq \emptyset$ then $t = t'$;*
2. *$\leq_N$ is a partial order and $\lfloor t \rfloor$ is finite for any $t \in T$;*
3. *the* initial marking *m is the set of $\leq_N$-minimal places;*
4. *$\#_N$ is irreflexive.*

With **ON** *we denote the full subcategory of* **PN** *having occurrence nets as objects.*

A *configuration* of an occurrence net N, formalising the intuitive idea of "concurrent run", is a subset $C \subseteq T$ such that C is left-closed w.r.t. $\leq_N$ and free of

conflicts. A set of places $X \subseteq S$ is called *concurrent*, written $conc(X)$, if $\lfloor X \rfloor$ is a configuration and $\neg(s <_N s')$ for all $s, s' \in X$.

We are now ready to define the unfolding of a Petri net. The unfolding construction unwinds a given net N into an occurrence net, starting from the initial marking, firing transitions in all possible ways and recording the corresponding occurrences. For the sake of presentation we give an equational definition.

Definition 8 (unfolding). *Let N be a Petri net. Its unfolding $\mathcal{U}(N)$ and the* folding morphism $\tau_N = (\eta, \beta) : \mathcal{U}(N) \to N$ *are the occurrence net and net morphism determined by the following recursive equations, where the components of the unfolding are primed:*

$$m' = \{\langle \emptyset, s\rangle \mid s \in m\}$$
$$S' = m' \cup \{\langle \{t'\}, s\rangle \mid t' = \langle X, t\rangle \in T' \ \wedge \ s \in t^\bullet\}$$
$$T' = \{\langle X, t\rangle \mid X \subseteq S' \ \wedge \ conc(X) \ \wedge \ t \in T \ \wedge \ \mu\beta(X) = {}^\bullet t\}$$

For $t' = \langle X, t\rangle \in T'$: $\quad {}^\bullet t' = X \quad$ *and* $\quad t'^\bullet = \{\langle \{t'\}, s\rangle \mid s \in t^\bullet\}$

$$\eta(t') = t \quad \text{iff} \ \ t' = \langle X, t\rangle$$
$$\beta(s', s) = 1 \ \ \text{iff} \ \ s' = \langle x, s\rangle \quad (x \in \mathbf{2}^{T'})$$

Observe that items in the unfolding are enriched with their causal histories. Place $s' = \langle x, s\rangle$ records its generator x (x is empty when the place is in the initial state, otherwise x is a singleton) and the place s in the original net; transition $t' = \langle X, t\rangle$ represents a firing of t that consumes the resources in X.

Proposition 9 (right adjoint [20, 13]). *The construction $\mathcal{U}$ extends to a functor $\mathcal{U} : \mathbf{PN} \to \mathbf{ON}$, right adjoint to the inclusion of* **ON** *into* **PN**.

Right adjoints preserve limits (see [12, 1]) and hence also pullbacks, which are special limits. As a consequence the unfolding of a pullback in **PN** is the pullback (in **ON**) of the unfoldings of the component nets. More precisely, given a pullback $\tau'_i : N_3 \to N_i$, $\tau_i : N_i \to N_0$ ($i \in \{1, 2\}$) in **PN**, we have that $\mathcal{U}(\tau'_i) : \mathcal{U}(N_3) \to \mathcal{U}(N_i)$, $\mathcal{U}(\tau_i) : \mathcal{U}(N_i) \to \mathcal{U}(N_0)$ ($i \in \{1, 2\}$) is a pullback in **ON**. This result will play a central role in the rest of this paper.

Example: Unfolding the nets N_0, N_1, N_2 and N_3 of Fig. 2 we obtain the occurrence nets O_0, O_1, O_2 and O_3 in Fig. 3 (ignore the shaded places and transitions for the moment). Transitions in the occurrence nets are named by using their label, with an additional index. The morphisms to the original nets are the obvious ones suggested by the labelling. By the considerations above, the occurrence net O_3 arising as unfolding of N_3 is the pullback of O_1 and O_2 along O_0.

The aim of this paper is to compute—in a distributed way—the projection of $\mathcal{U}(N_3)$ to $\mathcal{U}(N_i)$, i.e., the local view of component N_i, when taking into account the behaviour of the other component. The intuitive idea of local view is formalised by using factorisations.

It can be shown that, taking projections and embeddings as in Definition 4, the category **ON** is *pe*-structured. The only delicate point is to show that given an

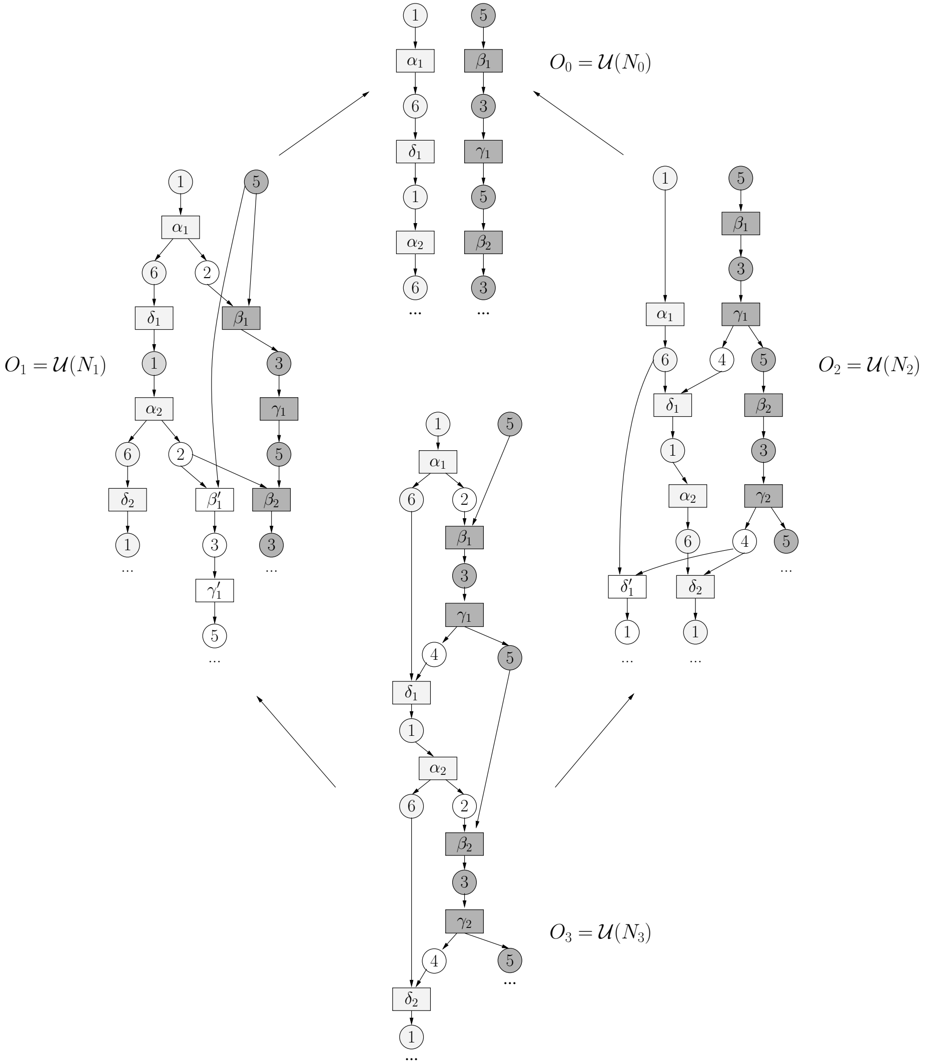


Fig. 3. Composition of unfoldings as pullback of occurrence nets

occurrence net morphism $\tau : O_1 \to O_2$, the net $\tau(O_1)$ as defined in Section 3 is a well-defined occurrence net, but this follows from the fact that occurrence net morphisms reflect causal chains (see [20], Lemma 3.3.6).

Definition 10 (projection of occurrence nets). *Let* $\tau = (\eta, \beta)\colon O_1 \to O_2$ *be an* **ON** *morphism, and let* $\tau^p\colon O_1 \to O_2^1$, $\tau^e\colon O_2^1 \to O_2$ *be the* pe*-factorisation of* τ. *Then the occurrence net* O_2^1 *is called the* projection *of* O_1 *onto* O_2.

Example: Consider the unfoldings of our running example in Fig. 3. The shaded places and transitions in O_0, O_1 and O_2 identify the projections O_0^3, O_1^3, O_2^3.

Transitions in O_1 and O_2 which disappear in the projection intuitively represent events that are infeasible if the net components interact. For instance, consider transition β_1' in O_1. From the point of view of N_1, transition δ_1 is a cause for β_1'. However, through the interface, transition β_1' in N_1 corresponds to β_1 in N_2, and in this latter net β_1 is a cause for δ_1. Hence β_1' turns out to be not firable.

5 Interleaving Structures and Their Properties

In order to be able to compute the local projection of the unfolding, intuitively, each net component needs to know the behavioural constraints on the events of the interface net imposed by the other component. Unfortunately, the idea of simply representing dependencies between events, i.e., causality and conflict, with prime event structures, and projecting to the interface the additional dependencies derived in each net component does not work. Consider, for instance, the occurrence nets in Fig. 4, where morphisms φ_i map any transition in N_i to the only transition in the interface net with the same label. Since the two γ-labelled transitions in N_1 are fused in N_0, the projection of causalities in N_1 to N_0 would result in an or-causality between $\{t_0, t_1\}$ and t_2, a phenomenon that is not expressible in a prime event structure. Still, from N_2 we obtain the information that t_2 must be fired before t_1. By combining this knowledge we discover that $t_0 < t_1 < t_2$ is the only possible order in which the transitions of N_0 can be executed. It can be shown that similar problems arise when considering more general partial order models including Winskel's general event structures [20].

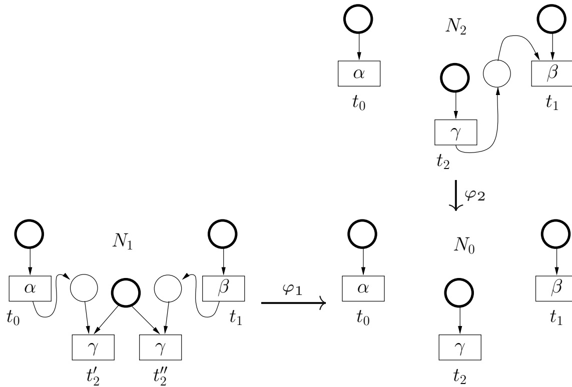


Fig. 4. Projecting dependency relations over the interface

The search for structures suitable to express possible orderings of events and forming a category with nice factorisation properties leads us to so-called interleaving structures. As their name suggests, these structures do not rely on partial orders, but, as discussed in [2], for practical purposes an efficient partial order representation based on occurrence nets can be devised.

Interleavings. For a set A, denote by A^* the set of finite sequences of elements of A and by $A^{\odot}$ the subset of sequences in A^* in which each element occurs at most once. A (partial) function $f : A \dashrightarrow B$ induces a function $f : A^* \to B^*$ (still denoted by f), where for $r \in A^*$ its image $f(r)$ is defined pointwise, removing from the sequence the elements on which f is undefined.

Definition 11 (interleaving structures). *An* interleaving structure *is a tuple $\mathcal{I} = (E, R, \lambda)$ 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 components of an interleaving structure $\mathcal{I}$ will be denoted by $E_{\mathcal{I}}$, $R_{\mathcal{I}}$, $\lambda_{\mathcal{I}}$, whereas the components of $\mathcal{I}_i$ will also be denoted by E_i, R_i, λ_i.

Definition 12 (interleaving morphisms). *Let $\mathcal{I}_i = (E_i, R_i, \lambda_i)$ with $i \in \{1,2\}$ be interleaving structures. An* interleaving morphism *from $\mathcal{I}_1$ to $\mathcal{I}_2$ is a partial function $\theta\colon E_1 \dashrightarrow E_2$ on events such that (i) $\lambda_2(\theta(e)) = \lambda_1(e)$ whenever $\theta(e)\downarrow$, and (ii) for every $r \in R_1$ it holds that $\theta(r) \in R_2$. We denote the category of interleaving structures and interleaving morphisms by* **Ilv**.

By Condition (2) above, θ must be injective on the events occurring in any single run. Pullbacks can be constructed in a quite straightforward way in **Ilv**.

Proposition 13 (pullbacks in Ilv). *Let $\theta_i\colon\mathcal{I}_i \to \mathcal{I}_0$, $i \in \{1,2\}$ be two interleaving morphisms. Their pullback in* **Ilv**, *denoted by $\pi_i\colon\mathcal{I}_3 \to \mathcal{I}_i$, $i \in \{1,2\}$ can be constructed as follows:*

- *Define E_3' as the pullback in the category of labelled sets and partial functions, and let $\pi_i'\colon E_3 \to E_i$ be the standard partial projections.*
- *Define $R_3 = \{r \in (E_3')^{\odot} \mid \pi_1(r) \in R_1 \wedge \pi_2(r) \in R_2\}$.*
- *Let $E_3 \subseteq E_3'$ be the subset of events in E_3' that occur in at least one run in R_3. Furthermore let $\pi_i = \pi_i'|_{E_3}\colon E_3 \to E_i$ be the projections restricted to E_3.*
- *Finally set $\lambda_3((e_1,e_2)) = \lambda_1(e_1) = \lambda_2(e_2)$, $\lambda_3((e_1,*)) = \lambda_1(e_1)$ and $\lambda_3((*,e_2)) = \lambda_2(e_2)$ for all events in E_3.*

Then $\mathcal{I}_3 = (E_3, R_3, \lambda_3)$ is the pullback object.

Factorisation Structures. We next show how to obtain a factorisation structure for **Ilv**. This is needed in order to project information about possible interleavings of events from each component down to the interface, where it can be read by the other component.

Definition 14 (projection, embedding). *An interleaving morphism $\theta\colon\mathcal{I}_1 \to \mathcal{I}_2$ is called* projection *if the induced function on runs $\theta\colon R_1 \to R_2$ is surjective. Morphism θ is called* embedding *if the mapping on events is a total injection.*

Observe that by definition any projection θ is surjective on the set of events.

Given any morphism $\theta\colon\mathcal{I}_1 \to \mathcal{I}_2$ in **Ilv**, a projection-embedding factorisation $\mathcal{I}_1 \xrightarrow{\theta^p} \mathcal{I}_2^1 \xrightarrow{\theta^e} \mathcal{I}_2$ can be obtained by taking as the runs of $\mathcal{I}_2^1$ all runs in $\mathcal{I}_2$ having a preimage under θ, and defining the set of events of $\mathcal{I}_2^1$ and θ^p, θ^e appropriately. The interleaving structure $\mathcal{I}_2^1$ is also called *projection* of $\mathcal{I}_1$ to $\mathcal{I}_2$ via θ.

Proposition 15 (Ilv (E, M)-structured). *The category* **Ilv** *is (E, M)-structured where E is the set of projections and M is the set of embeddings.*

It can be shown that, not only the embeddings, but also the projections are stable under pullbacks in **Ilv**. Note that an analogous proposition does not hold in **ON**. This is one of the reasons for resorting to interleaving structures.

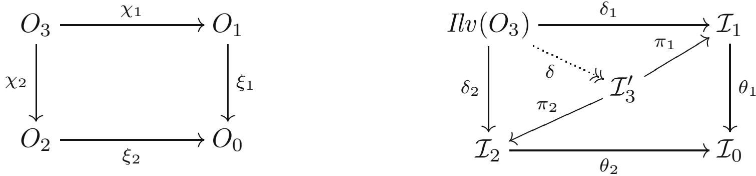
Projections of Interleaving Structures and Occurrence Nets. Every occurrence net O can be associated with an interleaving structure $Ilv(O)$ whose set of events coincides with the set of transitions of the net.

Definition 16. *Let $O = (S, T, \lambda, {}^\bullet(), ()^\bullet, m)$ be an occurrence net. Its interleaving structure is $Ilv(O) = (T, R, \lambda)$, where R consists of all runs $r \in T^\odot$ such that for every prefix r' of r the events occurring in r' form a configuration of O.*

In the following an element $r \in R_{Ilv(O)}$ will be called a run of O.

The mapping Ilv can be extended to a functor $Ilv : \mathbf{ON} \to \mathbf{Ilv}$. It can be seen that Ilv does not preserve pullbacks, but still a useful relation can be established between pullbacks in **ON** and in **Ilv**.

Lemma 17. *Consider a pullback diagram in* **ON** *as shown in the left-hand side below and take its image through the Ilv functor, thus obtaining the outer square in the right-hand diagram below. Furthermore let $\mathcal{I}_3'$ be the pullback in* **Ilv** *of θ_1 and θ_2. Then the mediating morphism $\delta: Ilv(O_3) \to \mathcal{I}_3'$ is a projection.*

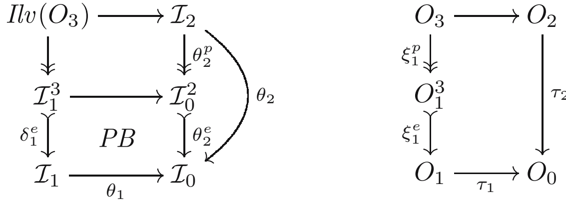


Summing up, we obtain a procedure for determining the projection of a pullback object in **ON** without actually constructing the pullback.

Proposition 18. *Let $\tau_i: O_i \to O_0$, $i \in \{1, 2\}$ be two occurrence net morphisms and let $\xi_i: O_3 \to O_i$, $i \in \{1, 2\}$ be their pullback. Then the projection O_1^3 and the morphism $O_1^3 \to O_1$ can be determined (without computing O_3) as follows:*

- *Determine the interleaving structures $\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2$ corresponding to O_0, O_1, O_2, i.e., $\mathcal{I}_i = Ilv(O_i)$, including their morphisms $\theta_i = Ilv(\tau_i): \mathcal{I}_0 \to \mathcal{I}_i$, $i \in \{1, 2\}$.*
- *Compute the projection-embedding factorisation $\mathcal{I}_2 \stackrel{\theta_2^p}{\to} \mathcal{I}_0^2 \stackrel{\theta_2^e}{\to} \mathcal{I}_0$ of θ_2.*
- *Take the pullback of θ_1 and θ_2^e and obtain the morphism $\delta_1^e: \mathcal{I}_1^3 \to \mathcal{I}_1$.*
- *Now take the subnet of O_1 that contains the transitions in the image of δ_1^e.*

This gives the projection O_1^3 of O_3 to O_1 with morphism $\xi_1^e : O_1^3 \to O_1$.



6 An Algorithm for Distributed Unfolding

We can now present a distributed unfolding algorithm based on interleaving structures. The algorithm takes as input a pair of net morphisms $\tau_i: N_i \to N_0$, $i \in \{1, 2\}$ obtained by decomposing a Petri net N_3 as in Proposition 6. Then it builds, in a stepwise fashion, the remaining morphisms of the commuting diagram in Fig. 5, where O_i^j is the projection of $\mathcal{U}(N_j)$ over $\mathcal{U}(N_i)$. When $\xi_i = (\eta_i, \beta_i)$, we will sometimes write $\xi_i(t)$ instead of $\eta_i(t)$.

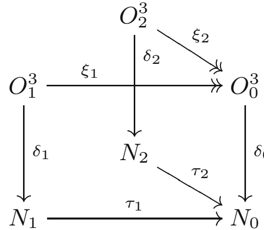


Fig. 5. Nets and morphisms involved in Algorithm 19

Algorithm 19 (distributed unfolding). Denote intermediate states of the occurrence nets and morphisms by $\bar{O}_0$, $\bar{O}_1$, $\bar{O}_2$, $\bar{\xi}_i$, $\bar{\delta}_j$. Start with occurrence nets corresponding to the initial places of N_0, N_1, N_2 and the appropriate corresponding morphisms. At any step transform the morphisms $\bar{\xi}_i$, $\bar{\delta}_i$ as follows: let $j \in \{1, 2\}$

(1) Look for a concurrent subset of places X in $\bar{O}_j$ such that $\bar{\delta}_j(X)$ is the pre-set of a transition t in N_j and furthermore[1]
    (*) there exists a run r of $\bar{O}_j$ that contains all causes of X and no consequences of X with $\bar{\xi}_j(r) \in \bar{\xi}_{3-j}(R_{Ilv(\bar{O}_{3-j})})$.
(2) Add $t' = \langle X, t\rangle$ with postset $\{\langle\{t'\}, s\rangle \mid s \in t^\bullet\}$ to $\bar{O}_j$;
    Update $\bar{\delta}_j$ by adding $t' \mapsto t$ and $\langle\{t'\}, s\rangle \mapsto s$.
(3) If $\tau_j(t) = t_0$ is defined,
    add a new transition $t'_0 = \langle\bar{\xi}_j(X), t_0\rangle$ with post-set $\{\langle\{t'_0\}, s_0\rangle \mid s_0 \in t_0{}^\bullet\}$ to $\bar{O}_0$, unless it is already present;
    Update $\bar{\delta}_0$ by adding $t'_0 \mapsto t_0$ and $\langle\{t'_0\}, s_0\rangle \mapsto s_0$;
    Update $\bar{\xi}_j$ by adding $t' \mapsto t'_0$ and $\langle\{t'\}, s\rangle \mapsto \langle\{t'_0\}, \tau_j(s)\rangle$.

[1] Condition (*) basically states that transition t can be fired after a run r of O_j and this run r is consistent with the behaviour of the other component. That is, there is a way to synchronise r and some run of O_{3-j}.

Assuming that there are two unfolders and a third process which manages the interface information (i.e., which records the projections of the runs of both components) then the checking of Condition (*) and step (3) are performed by unfolder j together with the interface manager, whereas the remaining steps can be performed by unfolder j on its own. Hence communication between unfolders 1 and 2 is restricted to communication via the interface manager.

A transition t in the occurrence net $\bar{O}_0$ is called *valid* if it appears in one of the runs of $R = \bar{\xi}_1(R_{Ilv(\bar{O}_1)}) \cap \bar{\xi}_2(R_{Ilv(\bar{O}_2)})$. A transition t' of $\bar{O}_j$ for $j \in \{1,2\}$ is *valid* if $\bar{\xi}_j(t') \uparrow$ or there is a run rt' in $\bar{O}_j$ such that $\bar{\xi}_j(rt') \in R$. Note that the algorithm will never generate a transition having a non-valid cause. Furthermore transitions of $\bar{O}_0$ might at some point not be valid but become valid at a later stage when corresponding pre-images have been generated by both unfolders.

Example: The above algorithm, applied to our running example, produces the shaded subparts of the nets in Fig. 3. For instance transition β_1' will never be added to $\bar{O}_1$. This transition may follow the run $\alpha_1\delta_1\alpha_2$, but there is no run r in O_2 for which we have $\xi_2(r) = \alpha_1\delta_1\alpha_2 = \xi_1(\alpha_1\delta_1\alpha_2)$.

In order to ensure that every enabled transition will eventually be chosen, the algorithm unfolds breadth-first: the sets X computed in step (1) of one round have to be worked out completely before those from the next round.

Proposition 20 (correctness of distributed unfolding). *Let $\bar{O}_0$, $\bar{O}_1$ and $\bar{O}_2$ denote the (infinite) unions of the sequences of nets produced by the algorithm above. By restricting $\bar{O}_0$, $\bar{O}_1$, $\bar{O}_2$ to the valid transitions (and their pre- and post-sets plus the initial places), one obtains exactly the occurrence nets O_0^3, O_1^3, O_2^3, where O_i^j is the projection of $\mathcal{U}(N_j)$ over $\mathcal{U}(N_i)$.*

7 Conclusion

We have presented a distributed algorithm for Petri net unfoldings based on pullback decompositions, whose use allows to factor the global unfolding into local views. In fact, computation of the—potentially large—global unfolding of a distributed system is avoided; local supervisors develop their local views, guided by message exchange with their peers through an interface net. As a data structure for this communication, event structures would appear as a natural choice, but for all considered branches of event structures (e.g., prime, bundle, stable, general event structures) important properties concerning factorisations and projections were lacking. This difficulty has been overcome by introducing the category of interleaving structures, which has been shown to enjoy the needed properties. The investigation of partially ordered models and related categories for the correlation of local views is a theme for future investigation. Some results concerning partial order representations for interleaving structures can be found in [2].

We gave a distributed unfolding algorithm in the case of two peers interacting through an interface. This calls for a generalisation to an arbitrary number of peers and unfolders. If all components share the same interface, this generalisation is straightforward: we only have to replace pullbacks by so-called

wide pullbacks of diagrams with several arrows having a common target object. The case where, for instance, the system consists of three components, and the interface between components 1 and 2 is different from the interface between components 1 and 3 is not straightforward and represents a matter of future investigation.

The task we addressed is closely related to that of [4, 9], so the differences deserve to be pointed out. A first one resides in the notion of system factorisation: [4, 9] use a composition operation between Petri nets based on place fusion, so transition occurrences have to be communicated between components and a sophisticated label coding is used to determine the local effect of a transition. Our approach essentially relies on a composition operation along an explicit interface, formalised as a pullback in a suitable category of nets; in the pullback decomposition, transitions acting on shared places are necessarily shared themselves. This contributed to making the algorithm simpler and easier to understand. Moreover, moving from the (computationally hard) products of event structures used in [4, 9] to the pullback of interleaving structures (possibly computed through their partial order representation) can lead to a gain in efficiency for the algorithm. More generally, the fact that our approach is developed in a categorical setting suggests a way for adapting it to different computational models, e.g., variations of Petri nets or more expressive models, like graph transformation systems [18]. This will only require to verify that the needed properties are satisfied by the category of models at hand.

Finally, distributed unfolding is orthogonal to the parallelisation of Petri net unfoldings in [11]: that work parallelises the computation of the global unfolding to gain efficiency, while we strive to avoid that computation altogether.

Acknowledgements. We are grateful to Andrea Corradini and Eric Fabre for fruitful discussions on preliminary versions of this work.

References

1. J. Adamek, H. Herrlich, and G.E. Strecker. *Abstract and Concrete Categories - The Joy of Cats*. Wiley, 1990.
2. P. Baldan, S. Haar, and B. König. Distributed unfolding of petri nets. Technical Report CS-2006-1, Department of Computer Science, University Ca' Foscari of Venice, 2006.
3. A. Benveniste, E. Fabre, Claude Jard, and S. Haar. Diagnosis of asynchronous discrete event systems, a net unfolding approach. *IEEE Trans. on Automatic Control*, 48(5):714–727, 2003.
4. A. Benveniste, S. Haar, E. Fabre, and C. Jard. Distributed monitoring of concurrent and asynchronous systems. In *Proc. of CONCUR'03*, volume 2761 of *LNCS*, pages 1–26. Springer, 2003.
5. R. Boel and J. van Schuppen. Decentralized failure diagnosis for discrete event systems with costly communication between diagnosers. In *Proc. 6th Int. Workshop on Discrete event Systems (WODES)*, pages 175–181, 2002.
6. C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic, 1999.

7. E. Fabre. Factorization of unfoldings for distributed tile systems, part 1: Reduced interaction case. Technical Report 4829, INRIA, May 2003.
8. E. Fabre. Factorization of unfoldings for distributed tile systems, part 2: General case. Technical Report 5186, INRIA, May 2004.
9. E. Fabre, A. Benveniste, S. Haar, and C. Jard. Distributed monitoring of concurrent and asynchronous systems. *Discrete Event Dynamic Systems: theory and application*, 15(1):33–84, 2005.
10. S. Genc and S. Lafortune. Distributed Diagnosis of discrete-event systems using Petri net unfoldings. In W.M.P. van der Aalst and E. Best, editors, *Proc. of ICATPN 2003*, volume 2679 of *LNCS*, pages 316–336. Springer, 2003.
11. K. Heljanko, V. Khomenko, and M. Koutny. Parallelisation of the petri net unfolding algorithm. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 371–385. Springer, 2002.
12. S. Mac Lane. *Categories for the working mathematician*. Springer, 1971.
13. J. Meseguer, U. Montanari, and V. Sassone. Process versus unfolding semantics for Place/Transition Petri nets. *Theoret. Comp. Sci.*, 153(1-2):171–210, 1996.
14. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoret. Comp. Sci.*, 13:85–108, 1981.
15. W. Reisig. *Petri Nets. An Introduction*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1982.
16. S. L. Ricker and J. van Schuppen. Decentralized failure diagnosis with asynchronous communication between diagnosers,. In *Proc. of the European Control Conference*, 2001.
17. S.L. Ricker and K. Rudie. Distributed knowledge for communication in decentralized discrete-event systems. In *Proc. of the IEEE Conference on Decision and Control (CDC)*, 2001.
18. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.
19. M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Trans. on Automatic Control*, 40(9):1555–1575, 1995.
20. G. Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 325–392. Springer, 1987.

On the μ-Calculus Augmented with Sabotage

Philipp Rohde

RWTH Aachen, Informatik VII
rohde@informatik.rwth-aachen.de

Abstract. We study logics and games over dynamically changing structures. Van Benthem's sabotage modal logic consists of modal logic with a cross-model modality referring to submodels from which a transition has been removed. We add constructors for forming least and greatest monadic fixed-points to that logic and obtain the sabotage μ-calculus. We introduce backup parity games as an extension of standard parity games where in addition, both players are able to delete edges of the arena and to store, resp., restore the current appearance of the arena by use of a fixed number of registers. We show that these games serve as model checking games for the sabotage μ-calculus, even if the access to registers follows a stack discipline. The problem of solving the games with restricted register access turns out to be PSPACE-complete while the more general games without a limited access become EXPTIME-complete (for at least three registers).

1 Introduction

In the classical framework of logics and corresponding model checking games, one considers changes of system states or movements of agents within a system, but the underlying structure is assumed to be static. This motivates the study of more general specification formalisms where we can directly address temporal changes of structures. In this contribution, we focus on the deletion of objects. Applications are, for example, (1) computer networks where connections may break down; (2) car navigation systems that cope with roadworks and traffic jams; (3) representations of knowledge where an increase in knowledge corresponds to a removal of uncertainty relations; and (4) Euler's famous problem of Seven Bridges of Königsberg (edges are removed after they were traversed for the first time). An algorithmic task for these systems is, for example, the reachability of designated states.

Van Benthem [2] proposed a modal logic with a transition-deleting modality, called sabotage modal logic SML. The main limitation of modal logics is the lack of a mechanism for unbounded iteration or recursion. To overcome this, we augment SML with constructors for forming least and greatest monadic fixed-points, which yields the sabotage μ-calculus SL_μ. This logic is capable of expressing iterative properties like reachability or recurrence as well as basic changes of the underlying structure, namely, the deletion of transitions.

In Section 2, we define the sabotage μ-calculus SL_μ and repeat some known results about the modal fragment SML. In Section 3, we introduce backup parity

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 142–156, 2006.
© Springer-Verlag Berlin Heidelberg 2006

games as token-moving games between two players. Depending on the type of the current vertex, the owner can decide on the further direction, or he can delete edges, or the current appearance of the arena is stored, resp., restored by use of a fixed number of registers. As winning condition for infinite plays, we use the well-known parity condition. In order to keep the complexity of solving these games low, we additionally require that registers can only be accessed by following a stack discipline: New values stored in a higher register also overwrite the values of all lower registers and the restoring of edges out of a higher register also erases all values of lower registers. The restriction on the register access guarantees that these games can be solved in polynomial space with respect to the size of the arena (when the number of registers is fixed). We also show that the problem of solving the games without this limited access becomes EXPTIME-complete, even for games with three registers.

In Section 4, we show that the model checking problem for SL_μ can be reduced to the problem of solving a backup parity game with limited access (by a polynomial time reduction). In fact, the maximum number of nested fixed-points of the given formula gives the number of registers in the game, and the dependency order of inductive fixed-point constructions corresponds to the stack discipline of register access. We conclude with Section 5 by giving a summary of the presented results and stating some open questions.

Due to lack of space, proofs are omitted or only sketched. Full proofs can be found in [10–Chaps. 5–6].

2 Sabotage μ-Calculus

Recall that the μ-calculus L_μ is obtained by adding constructors for forming least and greatest monadic fixed-points to propositional modal logic [7]. We extend this logic to the sabotage μ-calculus SL_μ by adding a cross-model modality referring to submodels from which a transition has been removed. For convenience, we define the syntax of SL_μ in negation normal form. All results easily extend to the general case (with the restriction that bounded variables only occur positively).

In what follows, let Σ be a finite alphabet, Prop a finite set of unary predicate symbols, and $\mathrm{Var} = \{X, Y, \ldots\}$ a set of propositional variables.

Definition 1. *A* Kripke structure $\mathcal{K}$ *over* Prop *is a tuple* (S, Σ, R, L)*, where* S *is an (at most countable) set of states,* $R \subseteq S \times \Sigma \times S$ *is a transition relation, and* $L : S \to 2^{\mathrm{Prop}}$ *is a labeling function assigning sets of predicates to states. Its size is defined by* $|\mathcal{K}| := |S| + |R|$*. For a set* $E \subseteq R$*, we define the substructure* $\mathcal{K} \setminus E := (S, \Sigma, R \setminus E, L)$*.*

Definition 2. *Formulae of the* sabotage μ-calculus SL_μ *are inductively defined by the following grammar. For* $p \in \mathrm{Prop}$*,* $a \in \Sigma$*, and* $X \in \mathrm{Var}$*, let*

$$\varphi ::= \top \mid \bot \mid p \mid \neg p \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \Diamond_a \varphi \mid \Box_a \varphi \mid \Diamond\!\!\!\cdot_a \varphi \mid \boxdot_a \varphi \mid \mu X.\varphi \mid \nu X.\varphi \ .$$

The fragment of formulae without fixed-point operators is called sabotage modal logic SML. *Let* $\mathrm{Cl}(\varphi)$ *be the set of subformulae,* $\mathrm{Var}(\varphi)$ *the set of variables and* $\mathrm{Bd}(\varphi)$ *the set of bounded variables of* φ. *Without loss of generality, we only deal with well-named formulae where every variable is bounded at most once and free variables are distinct from bounded variables. For each* $X \in \mathrm{Bd}(\varphi)$, *there is a unique binding definition* $\mathrm{Df}_\varphi(X) \in \mathrm{Cl}(\varphi)$ *equal to* $\mu X.\psi$ *or* $\nu X.\psi$.

Let $\mathcal{K} = (S, \Sigma, R, L)$ *be a Kripke structure and* φ *an* SL_μ*-formula. A* valuation of φ in $\mathcal{K}$ *is a function* $\mathcal{V} : \mathrm{Var}(\varphi) \to 2^S$. *The semantics of* SL_μ *is defined as the set* $\|\varphi\|_\mathcal{V}^\mathcal{K}$ *of states in which* φ *is true:*

$$\begin{aligned}
\|\top\|_\mathcal{V}^\mathcal{K} &:= S, \quad \|p\|_\mathcal{V}^\mathcal{K} := \{s \in S \mid p \in L(s)\}, \\
\|X\|_\mathcal{V}^\mathcal{K} &:= \mathcal{V}(X), \quad \|\varphi_1 \vee \varphi_2\|_\mathcal{V}^\mathcal{K} := \|\varphi_1\|_\mathcal{V}^\mathcal{K} \cup \|\varphi_2\|_\mathcal{V}^\mathcal{K}, \\
\|\Diamond_a \varphi\|_\mathcal{V}^\mathcal{K} &:= \{s \in S \mid \exists s' \in S : (s, a, s') \in R \wedge s' \in \|\varphi\|_\mathcal{V}^\mathcal{K}\}, \\
\|\diamonddot_a \varphi\|_\mathcal{V}^\mathcal{K} &:= \{s \in S \mid \exists t, t' \in S : (t, a, t') \in R \wedge s \in \|\varphi\|_\mathcal{V}^{\mathcal{K} \setminus \{(t,a,t')\}}\}, \text{ and} \\
\|\mu X.\varphi\|_\mathcal{V}^\mathcal{K} &:= \bigcap \{A \subseteq S \mid \|\varphi\|_{\mathcal{V}[X:=A]}^\mathcal{K} \subseteq A\}.
\end{aligned}$$

The semantics for the other operators is defined dually. For convenience, we set $(\mathcal{K}, s, \mathcal{V}) \models \varphi$ *iff* $s \in \|\varphi\|_\mathcal{V}^\mathcal{K}$.

We need to justify the definition of fixed-point formulae. Let φ be an SL_μ-formula. Then the function $A \mapsto \|\varphi\|_{\mathcal{V}[X:=A]}^\mathcal{K}$ is monotone and thus, it has a unique least fixed-points by Knaster-Tarski, which is equal to $\|\mu X.\varphi\|_\mathcal{V}^\mathcal{K}$.

Remark 1. There is a fundamental difference between $\Diamond_a$ and $\diamonddot_a$ with respect to fixed-points. When least and greatest fixed-points are constructed inductively, then movements are passed to the next stage, while the deletion of transitions is 'encapsulated' within a stage: The deletion is always restored when we proceed to the next step. This is due to the fact that, if we have determined $F_i = \|\psi\|_{\mathcal{V}[X:=F_{i-1}]}^\mathcal{K}$ for some formula ψ with deletion modalities and proceed to F_{i+1}, then we calculate $\|\psi\|_{\mathcal{V}[X:=F_i]}^\mathcal{K}$ over $\mathcal{K}$ and not over the substructure that results from the deletion of transitions.

Before we turn to the investigation of SL_μ, we repeat some results about the fragment SML. We start with an example:

Example 1. Consider the SML-formula $\varphi := \Diamond_a \Diamond_a \top \wedge \boxdot_a \Box_a \bot$. It is easy to see that every model of φ has exactly one a-transition and that it is a loop at the origin. In particular, SML is not bisimulation-invariant.

In [8, 9], it was shown that the sabotage modality already strengthens modal logic in such a way that all nice model-theoretic properties and algorithmic complexities get lost. In fact, from the viewpoint of complexities, SML much more resembles first-order logic than modal logic (with the exception that the formula complexity remains in PTIME):

Theorem 1 ([8, 9]). *For every* SML*-formula φ, there is an effectively constructible equivalent* FO*-formula with a size polynomial in $|\varphi|$. The model checking problem for* SML *is* PSPACE*-complete. Further,* SML *lacks the finite model property and the satisfiability problem becomes undecidable.* □

We proceed with two examples dealing with fixed-points and sabotage.

Example 2. For a given Kripke structure $\mathcal{K} = (S, \Sigma, R, L)$ and state $s \in S$ let $\hat{\mathcal{K}}_s$ be the unraveling of $\mathcal{K}$ at s. For convenience, we assume a unary alphabet Σ. We say that $\hat{\mathcal{K}}_s$ *contains a perfect subtree* if there is a non-empty subtree of $\hat{\mathcal{K}}_s$ such that each path of this subtree contains infinitely many splitting points. Let $\varphi := \nu X.\mu Y.(\boxdot\Diamond X \vee \Diamond Y)$ and suppose that $R \neq \emptyset$. We claim that $(\mathcal{K}, s) \models \varphi$ iff $\hat{\mathcal{K}}_s$ contains a perfect subtree. The L_μ-formula $\mu Y.(\psi \vee \Diamond Y)$ expresses that there is a finite path to a state where ψ holds. The subformula $\boxdot\Diamond X$ guarantees that there are at least two successors of the current state that belong to (the interpretation of) X. Let $G \subseteq S$ be the outer, greatest fixed-point according to X. If $s \in G$, then G is a perfect subtree of $\hat{\mathcal{K}}_s$. Conversely, if $P \subseteq S$ witnesses a perfect subtree of $\hat{\mathcal{K}}_s$, then $P \subseteq G$ by construction. Since P is prefix-closed, it follows that $s \in P$ and hence also $s \in G$.

The next example shows that we can ensure an infinite 'depth' of models:

Example 3. Let $\psi := \Diamond_b \top \wedge \odot_b(\Box_b \bot \wedge \nu Y.(\Box_a(\Diamond_b \top \wedge Y)))$ and $\varphi := \nu X.(\Diamond_a X \wedge \psi)$. Let $\mathcal{K} = (S, \Sigma, R, L)$ be a Kripke structure. Suppose that $(\mathcal{K}, t) \models \psi$ for some $t \in S$. Then t has exactly one outgoing b-transition and if this b-transition is removed, then every state that is reachable from t by a non-empty path along a-transitions still has a b-successor (due to the greatest fixed-point according to Y). In particular, every state that is reachable from t by a non-empty a-path is distinct from t. Suppose now that $(\mathcal{K}, s) \models \varphi$. Due to the greatest fixed-point according to X, there is an infinite path $\pi = s_0 \xrightarrow{a} s_1 \xrightarrow{a} s_2 \xrightarrow{a} \ldots$ with $s_0 = s$ and $(\mathcal{K}, s_i) \models \psi$ for every $i \in \mathbb{N}$. It follows that $s_i \neq s_j$ for every $i < j$ and thus, π consists of infinitely many pairwise distinct elements.

The complexity of model checking SL_μ can be readily determined.

Lemma 1. *The model checking problem for* SL_μ *is* PSPACE*-complete. The model checking problem for* SL_μ *with a fixed formula is* PTIME*-complete (program complexity).*

Proof. The hardness of combined model checking follows from the fact that SML is a fragment of SL_μ. Further, we can extend the embedding of SML into FO (cf. Theorem 1) to an embedding of SL_μ into $\mathrm{LFP}_{\mathrm{mon}}$, the first-order logic with least fixed-points over monadic relations. The model checking problem of the latter logic is known to be PSPACE-complete [13].

Further, the program complexity of L_μ is PTIME-complete [3] and L_μ is a fragment of SL_μ. Again, we can embed SL_μ into (full) LFP as above and we obtain equivalent LFP-formulae that are polynomial with respect to the sizes of the SL_μ-formulae. Since the program complexity of LFP is known to be PTIME-complete [12], the statement follows. □

3 Backup Parity Games

Recall that parity games are closely related to L_μ: They serve as model checking games for L_μ [5] and conversely, the winning condition of a parity game can be expressed by an L_μ-formula [14]. Despite the aforementioned solution of the model checking problem for SL_μ via LFP, we want to define a model checking game for SL_μ in the style of parity games. The games are defined as token-moving games between two players (called 0 and 1). Depending on the type of the current vertex, the owner of the vertex can decide on the further direction, or he can delete edges, or the current arena is stored, resp., restored. We use the parity condition as winning condition for infinite plays. To obtain a lower complexity, we require that the storing and restoring operations follow a stack discipline: New values stored in a higher register also overwrite the values of all lower registers and the restoring of edges out of a higher register erases all values of lower registers.

Definition 3. *Let (V, E) be a graph. For $v \in V$, let vE be the set of E-successors of v. If vE is a singleton set, then* $\mathrm{scc}(v)$ *denotes its unique element. For a set $A \subseteq V$, let $AE := \bigcup_{v \in A} vE$ be the set of E-successors of elements in A. Finally, let $\mathrm{Out}(A) := \{(v, v') \in E \mid v \in A, v' \in V\}$ be the set of edges with sources in A. A* backup parity game *of index n with m registers, (n, m)-backup game for short, is given by $\mathcal{G} = (\mathcal{A}, v_{\mathrm{in}})$, where $\mathcal{A}$ is an arena and v_{in} is an initial vertex of $\mathcal{A}$. An* arena *is a labeled graph $\mathcal{A} = (V, E, \Delta, \Omega)$ where V is a non-empty, finite set of vertices that can be partitioned into the following sets: (1) movement vertices M_i of player i; (2) deletion vertices D_i of player i; (3) storing vertices S_j for $j \in [1, m]$; and (4) restoring vertices R_j for $j \in [1, m]$. In this case, we write $V = (M_i, D_i, S_j, R_j)$. Let $M := M_0 \cup M_1$ be the set of movement vertices and $D := D_0 \cup D_1$ the set of deletion vertices. For the edge relation $E \subseteq V \times V$, we require that $|vE| = 1$ for each $v \in V \setminus M$. Finally, $\Delta \subseteq D \times 2^{\mathrm{Out}(M)}$ is a deletion relation and $\Omega : V \to \{0 \dots n\}$ is a priority function.*

A position *of the game is an element of $V \times (2^E)^{m+1}$. The initial position is $(v_{\mathrm{in}}, E \dots E)$. Let $(v, Y, X_1 \dots X_m)$ be the current position. Depending on v, a legal successor position is defined as follows. Assume that $v \in M_i$ for $i \in \{0, 1\}$. If $vY = \emptyset$, then Player i has lost the play. Otherwise, Player i chooses $v' \in vY$ and the new position becomes $(v', Y, X_1 \dots X_m)$. Assume that $v \in D_i$ for $i \in \{0, 1\}$. If there is no Ξ with $(v, \Xi) \in \Delta$ and $\emptyset \neq \Xi \subseteq Y$, then Player i has lost the play. Otherwise, Player i chooses such a set Ξ and the new position becomes $(\mathrm{scc}(v), Y \setminus \Xi, X_1 \dots X_m)$. If $v \in S_j$ for $j \in [1, m]$, then the new position becomes $(\mathrm{scc}(v), Y, Y \dots Y, X_{j+1} \dots X_m)$. Finally, if $v \in R_j$ for $j \in [1, m]$, then the new position becomes $(\mathrm{scc}(v), X_j, X_j \dots X_j, X_{j+1} \dots X_m)$.*

If the game goes on infinitely and the greatest number appearing infinitely often in the sequence $\Omega(v_0)\Omega(v_1)\Omega(v_2)\dots$ is even, then Player 0 wins the play; otherwise Player 1 wins. Finally, the size of a game is defined as $|\mathcal{A}| := |V| + |E| + \sum_{v \in D} \sum_{\Xi : (v, \Xi) \in \Delta} |\Xi|$.

Remark 2. By definition, a player gets stuck if either the current vertex is a moving vertex, but it has no successors with respect to the current set of edges.

Or it is a deletion vertex, but an appropriate deletion is not possible. Further, one has $Y \subseteq X_1 \subseteq \ldots \subseteq X_m \subseteq E$ for any position $(v, Y, X_1 \ldots X_m)$ that is reachable from the initial position.

We say that a game is in *normal form*, if Δ is a function $\Delta : D \to \mathrm{Out}(M)$ (in which case we also write δ) and $\Omega(v) = 0$ for each $v \in V \setminus M$. It is straightforward to show that for every (n, m)-backup game there is an equivalent (n, m)-backup game in normal form with a size linear in the size of the original game. For backup games in normal form, player-based choices are only made at movement vertices while for all other vertices, the successor position is uniquely determined (provided that the play does not end). Note that it suffices to consider only n and m that are bounded by some term in $\mathcal{O}(|V|)$.

There is a straightforward transformation of backup parity games into standard parity games, but at the cost of an exponential blow-up of the arena. To this end, the current appearance of the arena and the content of registers are encoded within the vertices of the new game.

Lemma 2. *For any (n, m)-backup game $\mathcal{G} = (\mathcal{A}, v_{\mathrm{in}})$, there is an equivalent parity game $\mathcal{G}' = (\mathcal{A}', v'_{\mathrm{in}})$ of same index such that $|\mathcal{A}'| \in \mathcal{O}(|\mathcal{A}| \cdot 2^{(m+1)|E|})$, where E is the set of edges in $\mathcal{A}$.* □

In fact, due to the restricted access to registers, the size of an equivalent parity game can be improved to $\mathcal{O}(|\mathcal{A}| \cdot (m+2)^{|E|})$. Since parity games are determined [4], we immediately obtain that for any (n, m)-backup game over the arena (V, E, Δ, Ω), the set of positions $V \times (2^E)^{m+1}$ can be partitioned into the winning regions W_0 and W_1 such that Player τ has a positional winning strategy on W_τ. Note that positional strategies for backup games can be easily transformed into automaton strategies over the arena using deterministic Mealy automata.

We turn to the algorithmic complexity of the problem of solving backup games, that is, the problem of deciding whether Player 0 can win the game $\mathcal{G} = (\mathcal{A}, v_{\mathrm{in}})$ starting from the initial position $(v_{\mathrm{in}}, E \ldots E)$, no matter how Player 1 moves (E is the set of edges in $\mathcal{A}$).

Theorem 2. *For a fixed number of registers, the problem of solving backup games is* PSPACE*-complete.*

Proof (Sketch). In [8], it was shown that the so-called *sabotage game*, where one player moves along edges of a finite graph and the other player removes an arbitrary edge in each round, is PSPACE-hard when the reachability of designated vertices is considered as game objective. Since the sabotage game is a special backup game, it follows that the problem of solving backup games is PSPACE-hard, even when restricted to games without priorities and without registers.

Let $\mathcal{G} = (\mathcal{A}, v_{\mathrm{in}})$ be an (n, m)-backup game with $\mathcal{A} = (V, E, \delta, \Omega)$ and $V = (M_i, D_i, S_j, R_j)$. Without loss of generality, we assume that $\mathcal{G}$ is in normal form. In what follows, we show that it can be decided whether Player 0 wins the game from $(v_{\mathrm{in}}, E \ldots E)$ in a space polynomial with respect to $|\mathcal{A}|$. To this end, we sketch a recursive alternating procedure with a running time polynomial in $|\mathcal{A}|$

(for m fixed). By APTIME = PSPACE (cf. [1]), it follows that for a fixed number of registers, the problem of solving the games belongs to PSPACE.

The algorithm is called with the current position $(v, Y, X_1 \dots X_m)$ as parameter (among some other data that is explained below). If v is a movement vertex, but a sink, or v is a deletion vertex, but the demanded edge is not present, then the algorithm immediately stops. In this case, it accepts or rejects subject to the player to which vertex v belongs. In all other cases, a successor vertex v' is chosen and the procedure is recursively called with parameter $(v', Y', X'_1 \dots X'_m)$ depending on the type of v. If $v \in M_0$, then the successor v' is non-deterministically guessed. If $v \in M_1$, then v' is chosen universally. If v is a deletion, storing, or restoring vertex, then its successor v' is uniquely determined. The parameters $Y', X'_1 \dots X'_m$ are then chosen according to the update rules of positions.

Beside the current position, the algorithm remembers for each vertex from $V \setminus D$ whether it was already visited and if so, which was the highest priority since then. This information is partly reset whensoever vertices are visited that alter the set of edges or the value of registers. The memory is realized by the functions $\tau : M \to [-1, n]$ as well as $\sigma_j : S_j \to [-1, n]$ and $\rho_j : R_j \to [-1, n]$ for $j \in [1, m]$. A function value of -1 means that this vertex was not seen yet or that this information was reset in the meantime. A function value greater or equal 0 gives the highest priority since the last visit of the respective vertex. The functions are updated depending on the type of the current vertex v:

- $v \in M$: If $\tau(v) \geq 0$, then the algorithm terminates. Otherwise, $\tau(v)$ is set to be 0. The value of $\tau(w)$ for each w in the domain of τ is updated to $\Omega(v)$ if $0 \leq \tau(w) < \Omega(v)$. The functions $\sigma_1 \dots \sigma_m$ and $\rho_1 \dots \rho_m$ are updated analogously.
- $v \in D$: The entire function τ is reset.
- $v \in S_j$ for $j \in [1, m]$: If $Y = X_j$ and $\sigma_j(v) \geq 0$, then the algorithm terminates. Otherwise, the entire functions τ, $\sigma_1 \dots \sigma_{j-1}$ and $\rho_1 \dots \rho_{j-1}$ are reset. Additionally, if $Y \subsetneq X_j$, then the functions σ_j and ρ_j are also reset. Finally, the value $\sigma_j(v)$ is set to be 0.
- $v \in R_j$ for $j \in [1, m]$: If $\rho_j(v) \geq 0$, then the algorithm terminates. Otherwise, the entire functions τ, $\sigma_1 \dots \sigma_j$ and $\rho_1 \dots \rho_{j-1}$ are reset and the value $\rho_j(v)$ is set to be 0.

For the correctness of the algorithm, one shows that the following statements hold. First, each computation branch corresponds to an admissible prefix of a play. In fact, by the choice of parameters for recursive calls, the computation tree forms a complete prefix of a game tree according to a strategy of Player 0. Second, if the algorithm terminates, then one of two cases have occurred: Either the current position is a sink (with respect to movement or deletion) or the corresponding prefix of a play can be extended to a loop, where the highest priority of this loop is known to the algorithm. Note that in this case, by the update of the functions τ, $\sigma_1 \dots \sigma_m$ and $\rho_1 \dots \rho_m$, exactly the same position from $V \times (2^E)^{m+1}$ is repeated. Hence, the algorithm can decide the winner of the play that is constituted by infinitely many repetitions of the loop. By positional determinacy, a player wins the game iff he wins by moving always identical

at same positions. Therefore, the validation of loops suffices to determine the winner.

We use a binary encoding of the parameters. Then each call of the procedure takes a time polynomial in $\mathcal{A}$. Regarding the running time, one shows that each computation branch of the alternating algorithm terminates after at most $\mathcal{O}(|V|^{4m+2})$ calls. There are four properties of backup games that are responsible for termination and that play a key role for estimating the running time. (1) Without deletion, storing, or restoring, a position is repeated after at most $|V|$ steps. (2) The deletion of edges is a one-way process: Without restoring, deletion vertices may occur at most $|D|$ times. If some deletion vertex is visited for the second time, then the related edge is no longer available and the corresponding player loses. (3) The storing of data by overwriting a register value with a *properly* smaller set is also bounded: Without storing or restoring by accessing higher registers, proper storing cannot be carried out more often than the number of edges. (4) Due to the dependency order, the algorithm is allowed to forget all information regarding lower registers when data is stored or restored. It follows that the alternating algorithm accepts its initial input iff Player 0 has a winning strategy in the game starting from $(v_{\text{in}}, E \dots E)$. This concludes the proof. □

Next, we settle the complexity of backup parity games when we skip the restriction that storing and restoring has to a follow stack discipline.

Definition 4. *An* (n,m)-RAM game *is defined analogously to an* (n,m)*-backup game, but the update of game positions for storing and restoring vertices is modified as follows. Let* $(v, Y, X_1 \dots X_m)$ *be the current position in the game. If* $v \in S_j$ *for* $j \in [1,m]$*, then the new position becomes* $(\text{scc}(v), Y, X'_1 \dots X'_m)$*, where* $X'_i := Y$ *if* $i = j$ *and* $X'_i := X_i$ *otherwise. If* $v \in R_j$ *for* $j \in [1,m]$*, then the new position becomes* $(\text{scc}(v), X_j, X_1 \dots X_m)$*. Thus, registers are accessed independently of each other. The updates for the other vertices remain unchanged.*

Theorem 3. *For* $n \geq 1$ *and* $m \geq 3$*, the problem of solving* (n,m)*-RAM games is* EXPTIME*-complete.*

Proof (Sketch). Let $\mathcal{G} = (\mathcal{A}, v_{\text{in}})$ be an (n,m)-RAM game. We can use the same transformation of Lemma 2 to obtain an equivalent parity game $\mathcal{G}' = (\mathcal{A}', v'_{\text{in}})$ of index n such that $|\mathcal{A}'| \in \mathcal{O}(|\mathcal{A}| \cdot 2^{(m+1)|E|})$, where E is the set of edges in $\mathcal{A}$. By a result of Jurdziński [6], parity games can be solved in a time polynomial with respect to the size of the arena and exponential with respect to the index of the game. It follows that $\mathcal{G}$ can be solved in time exponential with respect to the size of the arena, the index of the game, and the number of registers. Since we can assume that $n, m \in \mathcal{O}(|\mathcal{A}|)$, it follows that the problem of solving (n,m)-RAM games belongs to EXPTIME.

To establish the EXPTIME-hardness, we give a reduction from a two-player game introduced by Stockmeyer and Chandra [11], which is called *block game* and which is known to be EXPTIME-hard. It consists of an undirected graph $\mathcal{A}$, where each edge is labeled by a, b, or c, together with two sets F_0 and

F_1 of winning vertices. The players move alternatingly. A position is a tuple (τ, N_0, N_1) where $\tau \in \{0,1\}$ signifies whose turn it is, and N_0, N_1 are disjoint sets of markers (i.e., sets of vertices) that belong to Player 0 and Player 1. Assume that $(0, N_0, N_1)$ is the current position. Player 0 chooses one of his markers from N_0 and an edge label $x \in \{a,b,c\}$. Then he moves the chosen marker to a new vertex along a finite, non-empty path subject to the following conditions: (1) all traversed edges are labeled by x, and (2) no passed vertex (including the last one) carries a marker of either player. Player 0 immediately wins if he places his chosen marker on a vertex in F_0. The moves of Player 1 are defined analogously. The players are not permitted to pass. In order to cover plays where never any marker of Player τ is placed on a vertex in F_τ, we agree that Player 1 wins every infinite play.

We present an equivalent $(1,3)$-RAM game $\mathcal{G}' = (\mathcal{A}', v_{\text{in}})$ for a given block game $\mathcal{G} = (\mathcal{A}, p_{\text{in}})$ with initial position p_{in} that can be computed in polynomial time with respect to $|\mathcal{A}|$. Let V be the set of vertices of $\mathcal{A}$. The arena $\mathcal{A}'$ consists of several copies of $\mathcal{A}$ and special components in between. The first register of $\mathcal{G}'$ always contains the edge set of the original arena $\mathcal{A}'$ and is used to guarantee a 'clean board' at the beginning of each round. Positions of $\mathcal{G}$ are encoded in the second register of $\mathcal{G}'$ (see below). The arena $\mathcal{A}'$ contains an initial part that encodes the position p_{in}. It follows a loop that simulates two successive moves in $\mathcal{G}$. Let p be the encoded position of $\mathcal{G}$ at the beginning of the loop. Without loss of generality, we assume that a turn of Player 0 is simulated first. By deletion of edges, Player 0 chooses a candidate p' for a successor position of p and its encoding is stored into the third register. Then it is verified whether p' is indeed a legal successor of p by alternatingly restoring the second and the third register and checking all conditions separately. Note that for RAM games, the restoring does not affect the value of the other registers. If the check was successful and p' is a winning position for Player 0 in the block game, then Player 0 also wins the RAM game. If p' is not winning, then the value of the third register is shifted to the second register (by restoring out of the third register and immediately storing into the second register). Afterwards, the same procedure is repeated, but now Player 1 is the one who chooses the candidate for a successor position. At the end of the loop, the second register contains the encoding of a position and it is again Player 0's turn.

Markers are simulated by sets of edges. Assume that the current position of $\mathcal{G}$ is $p = (0, N_0, N_1)$. Player 0's choice of a successor position p' is simulated as follows. First, the original set of edges is restored out of the first register. Second, both players propose the sets N_0' and N_1' by deleting $|V| - |N_i|$ edge sets each of which corresponds to a marker. The result is stored into register 3. Third, it is checked whether $N_1' = N_1$: If they differ, then Player 0 can choose some edge that is present in register 2, but not in register 3 and lead the play towards a sink of Player 1. And last, if $N_1' = N_1$, then it is verified that exactly one marker in N_0 was moved according to the rules of the block game. This is done by the following steps: (1) By entering a special component, Player 0 asserts that a marker at vertex $v \in V$ was moved; (2) it is checked whether v

carried a marker in N_0, but not in N_0'; (3) by entering a special component, Player 0 asserts that the marker at v was moved along a x-labeled path for some $x \in \{a, b, c\}$; (4) it is checked whether there is a non-empty x-labeled path from v to a vertex $w \in V$ such that no intermediate vertex carries a marker of either player: Player 0 chooses the edges that are traversed; if there is a marker from $N_0 \cup N_1$ at intermediate vertices, then Player 1 can lead the play to a sink of Player 0; (5) finally, it is checked whether no other marker than the one at v was moved (otherwise, Player 1 can lead the play to a sink of Player 0).

The priorities 0 and 1 are used to ensure that players do not move ad infinitum when they simulate the movement of markers. If the game does not end, then Player 1 wins, because priority 1 is visited infinitely often. □

Note that a RAM game with only one register necessarily follows a stack discipline and thus, it is already a backup game that can be solved in polynomial space. It remains open whether this is true for RAM games with two registers.

4 A Model Checking Game for SL_μ

In this section, we show that the model checking problem for SL_μ can be reduced to the problem of solving a backup game. We present a backup game $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ for a finite Kripke structure $\mathcal{K}$, an SL_μ-formula φ, and a valuation $\mathcal{V}$ such that for every state s: Player 0 wins the game from some designated vertex iff $(\mathcal{K}, s, \mathcal{V}) \models \varphi$. The construction is an adaptation of the one for L_μ, which is based on a transformation of the model checking problem for L_μ into the emptiness problem for parity tree automata [5].

In what follows, we fix a finite Kripke structure $\mathcal{K} = (S, \Sigma, R, L)$ and an SL_μ-formula φ over Σ, both over the set Prop of predicates symbols. Further, we assume that $\lambda \notin \Sigma$. Let $\mathcal{V} : \mathrm{Var}(\varphi) \to 2^S$ be a valuation for φ.

The structure $\mathcal{K}_\varphi = (S_\varphi, \Sigma \cup \{\lambda\}, R_\varphi, \emptyset)$ is induced by the structure of φ: First, there is a state for each subformula in $\mathrm{Cl}(\varphi)$. For simplicity, we name the states after subformulae. Second, there are two additional states s_X and r_X for each $X \in \mathrm{Bd}(\varphi)$. Third, we add an extra state q_a for each $a \in \Sigma$. The latter states are needed for deletion purposes and are independent of the structure of φ. Let $\mathrm{init} : \mathrm{Cl}(\varphi) \to S_\varphi$ be defined by $\mathrm{init}(\psi) := r_X$ if $\psi = X$ and $X \in \mathrm{Bd}(\varphi)$, $\mathrm{init}(\psi) := s_X$ if $\psi = \mu X.\psi'$ or $\psi = \nu X.\psi'$, and $\mathrm{init}(\psi) := \psi$ otherwise.

We define R_φ by giving a list of transitions for each type of subformula. Let $\psi \in \mathrm{Cl}(\varphi)$. (1) The states $\top$, $\bot$, p, $\neg p$, and X for $X \in \mathrm{Var}(\varphi) \setminus \mathrm{Bd}(\varphi)$ are sinks; (2) if $\psi = \psi_1 \vee \psi_2$ or $\psi = \psi_1 \wedge \psi_2$, then there are the transitions $\psi \xrightarrow{\lambda} \mathrm{init}(\psi_1)$ and $\psi \xrightarrow{\lambda} \mathrm{init}(\psi_2)$; (3) if $\psi = \Diamond_a \psi'$ or $\psi = \Box_a \psi'$ for $a \in \Sigma$, then there is the transition $\psi \xrightarrow{a} \mathrm{init}(\psi')$; (4) if $\psi = \diamondminus_a \psi'$ or $\psi = \boxminus_a \psi'$ for $a \in \Sigma$, then there is the transition $\psi \xrightarrow{\lambda} \mathrm{init}(\psi')$; (5) if $\psi = \mu X.\psi'$ or $\psi = \nu X.\psi'$, then there are the transitions $s_X \xrightarrow{\lambda} \psi$, $\psi \xrightarrow{\lambda} \mathrm{init}(\psi')$, $r_X \xrightarrow{\lambda} X$, and $X \xrightarrow{\lambda} \psi$; (6) there is a transition $q_a \xrightarrow{a} q_a$ for every $a \in \Sigma$.

Let $\mathcal{K} \otimes \mathcal{K}_\varphi$ be the synchronized product of $\mathcal{K}$ and $\mathcal{K}_\varphi$ where predicates are ignored: for $a \in \Sigma$, we have $(s, t) \xrightarrow{a} (s', t')$ iff $s \xrightarrow{a} s'$ in $\mathcal{K}$ and $t \xrightarrow{a} t'$ in $\mathcal{K}_\varphi$ as

well as $(s,t) \xrightarrow{\lambda} (s',t')$ iff $t \xrightarrow{\lambda} t'$ in $\mathcal{K}_\varphi$. Let $G = (V,E)$ be the transition graph of $\mathcal{K} \otimes \mathcal{K}_\varphi$ without transition labels. Let m be the fixed-point depth of φ. The game $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ has then m registers. We start with the declaration of the vertices in V as movement, deletion, storing, or restoring vertices. Each (s,q_a) for $s \in S$ and $a \in \Sigma$ belongs to M_0. Let $s \in S$, $\psi \in \mathrm{Cl}(\varphi)$, and $a \in \Sigma$. Then $(s,\psi) \in M_0$ if (1) $\psi = \bot$, $\psi = \psi_1 \vee \psi_2$, $\psi = \Diamond_a \psi'$, or $\psi = \nu X.\psi'$, or (2) $\psi = p$ and $p \notin L(s)$, or $\psi = \neg p$ and $p \in L(s)$, or (3) $\psi = X$ and $s \notin \mathcal{V}(X)$. The movement vertices M_1 of Player 1 are defined dually. We set $(s,\psi) \in D_0$ if $\psi = \diamond\!\!\!\cdot_a \psi'$ and $(s,\psi) \in D_1$ if $\psi = \boxdot_a \psi'$. Finally, let $\mathrm{fh}_\varphi(X)$ to be the maximum number of nested fixed-point operators in $\mathrm{Df}_\varphi(X)$ for $X \in \mathrm{Bd}(\varphi)$. Then we set $(s, s_X) \in S_{\mathrm{fh}_\varphi(X)}$ and $(s, r_X) \in R_{\mathrm{fh}_\varphi(X)}$ for $X \in \mathrm{Bd}(\varphi)$.

Next, we define the deletion relation Δ. For $t,t' \in S$ and $a \in \Sigma$, let $\Xi^\varphi_{t,a,t'}$ be the following set:

$$\{((t,\psi),(t',\mathrm{init}(\psi'))) \mid \psi \in \mathrm{Cl}(\varphi) \wedge (\psi = \Diamond_a \psi' \text{ or } \Box_a \psi')\} \cup \{((t,q_a),(t',q_a))\}.$$

Note that, if $(t,a,t') \in R$, then we have $\Xi^\varphi_{t,a,t'} \subseteq \mathrm{Out}(M)$. By the synchronized product, we have $((t,q_a),(t',q_a)) \in E$ iff $(t,a,t') \in R$. Thus, $\Xi^\varphi_{t,a,t'}$ is non-empty for $(t,a,t') \in R$, regardless of the structure of φ. For $a \in \Sigma$ and $\psi \in \mathrm{Cl}(\varphi)$ with $\psi = \diamond\!\!\!\cdot_a \psi'$ or $\psi = \boxdot_a \psi'$, we set for every $s \in S$: $((s,\psi),\Xi^\varphi_{t,a,t'}) \in \Delta$ iff $t,t' \in S \wedge (t,a,t') \in R$.

We conclude the definition of the arena $\mathcal{A}_{\mathcal{K},\varphi,\mathcal{V}}$ by specifying the priority function $\Omega : V \to \mathbb{N}$. We first define a function Ω' for $\mathcal{K}_\varphi$ and then we extend Ω' to V. For $X \in \mathrm{Bd}(\varphi)$, let $B_\varphi(X) := \mathrm{Bd}(\mathrm{Df}_\varphi(X)) \setminus \{X\} \subseteq \mathrm{Var}(\varphi)$. Then we define $\Omega' : S_\varphi \to \mathbb{N}$ by $\Omega'(s) := 0$ for every $s \in S_\varphi \setminus \mathrm{Bd}(\varphi)$, and $\Omega'(X) := \min\{c \in \mathbb{N} \mid c \text{ odd} \wedge c > \max\{0, \{\Omega'(Y) \mid Y \in B_\varphi(X)\}\}\}$ if X is a μ-variable of φ, and $\Omega'(X) := \min\{c \in \mathbb{N} \mid c \text{ even} \wedge c > \max\{0, \{\Omega'(Y) \mid Y \in B_\varphi(X)\}\}\}$ if X is a ν-variable of φ.

We set $\Omega((s,\psi)) = \Omega'(\psi)$ for every $s \in S$ and $\psi \in S_\varphi$. Finally, let $n := \max\{\Omega'(X) \mid X \in \mathrm{Bd}(\varphi)\}$. This concludes the definition of the arena $\mathcal{A}_{\mathcal{K},\varphi,\mathcal{V}} = (V,E,\Delta,\Omega)$ and the (n,m)-backup game $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$. Note that n and m depend on φ only. Regarding the size of the game, it is straightforward to check that $|\mathcal{A}_{\mathcal{K},\varphi,\mathcal{V}}|$ is quadratic with respect to $|\mathcal{K}| \cdot |\varphi|$.

Before we show that $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ indeed serves as a model checking game for SL_μ, we observe that the initial content of registers has no effect on plays starting 'at the top' of the game.

Lemma 3. *With the same notation as before, one has for each state $s \in S$ and for each $X_1 \dots X_m \in 2^E$ that Player τ wins $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ from position $((s,\mathrm{init}(\varphi)),E,$ $X_1 \dots X_m)$ iff he wins $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ from position $((s,\mathrm{init}(\varphi)),E,E \dots E)$.* □

We need two auxiliary results concerning backup games. The first one deals with winning regions of subgames. The second one provides an unfolding of the parity condition, which we need for the fixed-point operators. The unfolding is a classical construction based on Knaster-Tarski.

Lemma 4. *Let $\mathcal{A} = (V,E,\Delta,\Omega)$ be the arena of an (n,m)-backup game $\mathcal{G}$ with $V = (M_\tau, D_\tau, S_j, R_j)$. Suppose that $V' \subseteq V$ such that $V'E \subseteq V'$ and that*

for each $v \in V'$*, if* $(v, \Xi) \in \Delta$ *and* $\Xi \neq \emptyset$*, then* $\Xi \cap (V' \times V') \neq \emptyset$*. Let* $u \in V'$ *and* $X_0 \dots X_m \subseteq E$ *be fixed. We define* $V' = (M_\tau \cap V', D_\tau \cap V', S_j \cap V', R_j \cap V')$*,* $E' := E \cap (V' \times V')$*,* $X'_i := X_i \cap E'$ *for each* $i \in [0, m]$*, and* $\Delta' := \{(v, \Xi \cap E') \mid v \in V' \wedge (v, \Xi) \in \Delta\}$*. Let* $\mathcal{G}'$ *be the* (n, m)*-backup game with arena* $\mathcal{A}' = (V', E', \Delta', \Omega|_{V'})$*. Then Player* τ *wins* $\mathcal{G}$ *from* $(u, X_0 \dots X_m)$ *iff he wins* $\mathcal{G}'$ *from* $(u, X'_0 \dots X'_m)$*.* □

We turn to the unfolding of the parity condition. Let $\mathcal{A} = (V, E, \Delta, \Omega)$ be the arena of an (n, m)-backup game $\mathcal{G}$ with $V = (M_\tau, D_\tau, S_j, R_j)$. Let $\Omega_{\max} := \max_{v \in V} \Omega(v)$, $T := \Omega^{-1}(\Omega_{\max})$, $U := TE$ and $\kappa := |T| + 1$. We make the following assumptions: (1) $\Omega_{\max}$ is even; (2) $T \subseteq M$; (3) every $v \in T$ has a unique successor $\text{scc}(v)$; (4) for every $u \in U$ and every play π that starts from $(u, E \dots E)$, if $\pi_i = (v, X_0 \dots X_m)$ for some $v \in T$, then $X_j = E$ for every $j \in [0, m]$; (5) if $(v, \Xi) \in \Delta$ for some $v \in V$, then $\Xi \cap (T \times U) = \emptyset$.

Under this assumptions, the *unfolding* of $\mathcal{G}$ is a sequence of (n, m)-backup games $\mathcal{G}^i$ for $i \in [0, \kappa]$. Let $E^- := E \setminus (T \times U)$ and $\mathcal{A}^- := (V, E^-, \Delta, \Omega)$. Note that the vertices in T become terminal in $\mathcal{A}^-$. The arena of $\mathcal{G}^i$ coincides with $\mathcal{A}^-$ up to the winning condition for T. For every $i \in [0, \kappa]$, we define a decomposition $T = T_0^i \cup T_1^i$ and declare Player τ to be the winner of the game $\mathcal{G}^i$ when a play reaches $v \in T_\tau^i$. Let $W_\tau^i \subseteq V \times (2^{E^-})^{m+1}$ be the winning region of Player τ in the game $\mathcal{G}^i$. Clearly, W_τ^i depends on the decomposition $T = T_0^i \cup T_1^i$. In turn, the decomposition of T for $i + 1$ depends on W_τ^i: We define $T_0^0 := T$ and $T_0^{i+1} := \{v \in T \mid (\text{scc}(v), E^- \dots E^-) \in W_0^i\}$.

It is easy to check that $T_1^0 \subseteq T_1^1 \subseteq \dots \subseteq T_1^\kappa$ and $W_1^0 \subseteq W_1^1 \subseteq \dots \subseteq W_1^\kappa$. By determinacy, we also have $T_0^0 \supseteq T_0^1 \supseteq \dots \supseteq T_0^\kappa$ and $W_0^0 \supseteq W_0^1 \supseteq \dots \supseteq W_0^\kappa$. Since $\kappa = |T| + 1$ and $T_1^i \subseteq T$ for each $i \in [0, \kappa]$, there exists $\alpha < \kappa$ such that $T_1^\alpha = T_1^{\alpha+1}$ (and then also $T_0^\alpha = T_0^{\alpha+1}$, $W_\tau^\alpha = W_\tau^{\alpha+1}$). We claim that we can determine the winner of a play in the original game $\mathcal{G}$ that starts from a vertex $u \in U$ by considering this fixed-point of winning regions for the unfolding of $\mathcal{G}$:

Lemma 5. *Let* $\mathcal{G}$*,* U*, and* κ *be as before and let* $\mathcal{G}^0 \dots \mathcal{G}^\kappa$ *be the unfolding of* $\mathcal{G}$*. Then for every* $u \in U$*, Player* τ *wins* $\mathcal{G}$ *from* $(u, E \dots E)$ *iff he wins* $\mathcal{G}^\kappa$ *from* $(u, E^- \dots E^-)$*.* □

If $\mathcal{G}$ is as before, but $\Omega_{\max}$ is odd, then we can proceed to the dual game where the roles of the players are swapped and the priority function is increased by one. We are now prepared to prove the main result of this section:

Theorem 4. *Suppose that* $\mathcal{K} = (S, \Sigma, R, L)$ *is a finite Kripke structure with state* $s \in S$*,* φ *is an* SL_μ*-formula over* Σ*, and* $\mathcal{V} : \text{Var}(\varphi) \to 2^S$ *is a valuation. Then* $(\mathcal{K}, s, \mathcal{V}) \models \varphi$ *iff Player* 0 *wins* $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ *from* $((s, \text{init}(\varphi)), E \dots E)$*, where* E *is the edge relation of the arena of* $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$*.*

Proof (Sketch). The proof is by induction on the structure of φ. We only present some interesting cases. Case $\varphi = \Diamond_a \psi$. Let $\mathcal{A} = (V, E, \Delta, \Omega)$ be the arena of $\mathcal{G} := \mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ and $\mathcal{A}' = (V', E', \Delta', \Omega')$ be the arena of $\mathcal{G}' := \mathcal{G}_{\mathcal{K},\psi,\mathcal{V}}$. It is easy to see that $\mathcal{G}'$ is a subgame of $\mathcal{G}$ that meets the requirements of Lemma 4. It follows that for each $t \in S$, Player 0 wins $\mathcal{G}$ from $((t, \text{init}(\psi)), E \dots E)$ iff he

wins $\mathcal{G}'$ from $((t, \mathrm{init}(\psi)), E' \dots E')$. We have $\mathrm{init}(\varphi) = \varphi$ and $(s, \varphi) \in M_0$. By definition of $\mathcal{A}$, there is an edge from (s, φ) to $(t, \mathrm{init}(\psi))$ iff $(s, a, t) \in R$. Thus

$$\begin{aligned}
&\text{Player 0 wins } \mathcal{G} \text{ from } ((s,\varphi), E \dots E) \\
\iff &\exists t \in S : (s,a,t) \in R \text{ and Player 0 wins } \mathcal{G} \text{ from } ((t,\mathrm{init}(\psi)), E \dots E) \\
\iff &\exists t \in S : (s,a,t) \in R \text{ and Player 0 wins } \mathcal{G}' \text{ from } ((t,\mathrm{init}(\psi)), E' \dots E') \\
\iff &\exists t \in S : (s,a,t) \in R \text{ and } (\mathcal{K}, t, \mathcal{V}) \models \psi \quad \text{[by induction]} \\
\iff &(\mathcal{K}, s, \mathcal{V}) \models \varphi.
\end{aligned}$$

Case $\varphi = \Diamond\!\!\!\cdot_a \psi$. Let $\mathcal{A} = (V, E, \Delta, \Omega)$ be the arena of $\mathcal{G} := \mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$. We have $\mathrm{init}(\varphi) = \varphi$, $(s, \varphi) \in D_0$, and (s, φ) has the unique E-successor $(s, \mathrm{init}(\psi))$. By definition, there is $((s, \varphi), \Xi) \in \Delta$ with $\emptyset \neq \Xi \subseteq E$ iff there are $t, t' \in S$ with $(t, a, t') \in R$ and $\Xi = \Xi^{\varphi}_{t,a,t'}$. Thus, it follows that Player 0 wins $\mathcal{G}$ from $((s, \varphi), E \dots E)$ iff there are $t, t' \in S$ with $(t, a, t') \in R$ such that Player 0 wins $\mathcal{G}$ from $((s, \mathrm{init}(\psi)), E \setminus \Xi^{\varphi}_{t,a,t'}, E \dots E)$. Let $E^- := E \setminus \Xi^{\varphi}_{t,a,t'}$. By Lemma 3, we have that Player 0 wins $\mathcal{G}$ from $((s, \mathrm{init}(\psi)), E^-, E \dots E)$ iff he wins $\mathcal{G}$ from $((s, \mathrm{init}(\psi)), E^-, E^- \dots E^-)$.

The arena (V, E^-, Δ, Ω) is identical with the arena $\mathcal{A}' := (V', E', \Delta', \Omega')$ of the game $\mathcal{G}' := \mathcal{G}_{\mathcal{K}\setminus\{(t,a,t')\},\varphi,\mathcal{V}}$ up to the deletion relation Δ'. In particular, we have $E^- = E'$. But for any play in $\mathcal{G}$ starting at $((s, \mathrm{init}(\psi)), E^- \dots E^-)$, neither player can choose $\Xi^{\varphi}_{t,a,t'}$ at deletion vertices without losing immediately. It follows that Player 0 wins $\mathcal{G}$ from $((s, \mathrm{init}(\psi)), E^- \dots E^-)$ iff he wins $\mathcal{G}'$ from $((s, \mathrm{init}(\psi)), E' \dots E')$. Finally, let $\mathcal{G}'' := \mathcal{G}_{\mathcal{K}\setminus\{(t,a,t')\},\psi,\mathcal{V}}$ with arena $\mathcal{A}'' = (V'', E'', \Delta'', \Omega'')$. Again, $\mathcal{G}''$ is a subgame of $\mathcal{G}'$ that contains the state $(s, \mathrm{init}(\psi))$ and that meets the requirements of Lemma 4. It follows that Player 0 wins $\mathcal{G}'$ from $((s, \mathrm{init}(\psi)), E' \dots E')$ iff he wins $\mathcal{G}''$ from $((s, \mathrm{init}(\psi)), E'' \dots E'')$. By induction, this is equivalent to $(\mathcal{K}\setminus\{(t, a, t')\}, s, \mathcal{V}) \models \psi$. Together, we get that Player 0 wins $\mathcal{G}$ from $((s, \varphi), E \dots E)$ iff there exists $t, t' \in S$ with $(t, a, t') \in R$ and $(\mathcal{K} \setminus \{(t, a, t')\}, s, \mathcal{V}) \models \psi$. The latter is equivalent to $(\mathcal{K}, s, \mathcal{V}) \models \varphi$.

Case $\varphi = \nu X.\psi$. Let $\mathcal{A} = (V, E, \Delta, \Omega)$ be the arena of the game $\mathcal{G} := \mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ with $V = (M_\tau, D_\tau, S_j, R_j)$. By definition, we have that $\Omega_{\max} = \Omega'(X)$ is even. Let $T := \Omega^{-1}(\Omega_{\max})$. By definition of $\mathcal{A}$, we have $T = \{(t, X) \mid t \in S\} \subseteq M$, $|(t, X)E| = 1$ for each $t \in S$, and $U := TE = \{(t, \varphi) \mid t \in S\}$. Let $\kappa := |T| + 1 = |S| + 1$. We have $\mathrm{fh}_\varphi(X) = m$ and thus, register $\mathrm{fh}_\varphi(X)$ is the highest register of $\mathcal{G}$. Further, we have $S_m = \{(t, \mathrm{init}(\varphi)) \mid t \in S\}$ and each of these vertices has a single E-successor, namely (t, φ), and no incoming E-edge. In particular, Player 0 wins $\mathcal{G}$ from $((s, \mathrm{init}(\varphi)), E \dots E)$ iff he wins $\mathcal{G}$ from $((s, \varphi), E \dots E)$. Every vertex (t, X) occurs in combination with the restoring vertex (t, r_X). Let $t, t' \in S$ and π be a play that starts from $((t, \varphi), E \dots E)$ and that reaches some $\pi_i = ((t', X), X_0 \dots X_m)$. Note that no storing vertex from S_m can occur in π. Thus, we have $X_m = E$ and π_{i-1} visits vertex (t', r_X). Be definition of the restoring process, it follows that $X_j = E$ for every $j \in [0, m]$. Finally, the edges $(t, X) \to (t, \varphi)$ for $t \in S$ do not occur in Δ. It follows that $\mathcal{G}$ meets the requirements of Lemma 5.

Before we proceed, we fix some notation. Let $\mathcal{W} : \text{Var}(\varphi) \to 2^S$ be a valuation of φ. Note that the definitions of the edge relation, the deletion relation, and the priority function of a game $\mathcal{G}_{\mathcal{K},\chi,\mathcal{W}}$ do not depend on the valuation $\mathcal{W}$. Let $\mathcal{A}_\mathcal{W} = (V_\mathcal{W}, E, \Delta, \Omega)$ be the arena of $\mathcal{G}_\mathcal{W} := \mathcal{G}_{\mathcal{K},\varphi,\mathcal{W}}$. The game $\mathcal{G}_\mathcal{W}$ is identical to $\mathcal{G}$ up to the assignment of vertices (t, Y) for $t \in S, Y \in \text{Var}(\varphi)$ to one of the players. Let $E^- := E \setminus (T \times U)$ and $\mathcal{G}^-_\mathcal{W}$ be the game with arena $(V_\mathcal{W}, E^-, \Delta, \Omega)$. Finally, we define the game $\mathcal{G}^*_\mathcal{W} := \mathcal{G}_{\mathcal{K},\psi,\mathcal{W}}$ with arena $\mathcal{A}^*_\mathcal{W} = (V^*_\mathcal{W}, E^*, \Delta^*, \Omega^*)$. The variable X occurs free in ψ. Thus, the vertices (t, X) for $t \in S$ have no outgoing edges in $\mathcal{A}^*_\mathcal{W}$.

The game $\mathcal{G}^*_\mathcal{W}$ is a subgame of $\mathcal{G}^-_\mathcal{W}$ that contains the vertices $(t, \text{init}(\psi))$ for every $t \in S$ and that meets the requirements of Lemma 4. In $\mathcal{A}^-_\mathcal{W}$, each vertex (t, φ) for $t \in S$ has the unique successor $(t, \text{init}(\psi))$. Together with the induction hypothesis, it follows for every $t \in S$ that Player 0 wins $\mathcal{G}^-_\mathcal{W}$ from $((t, \varphi), E^- \dots E^-)$ iff he wins $\mathcal{G}^-_\mathcal{W}$ from $((t, \text{init}(\psi)), E^- \dots E^-)$ iff he wins $\mathcal{G}^*_\mathcal{W}$ from $((t, \text{init}(\psi)), E^* \dots E^*)$ iff $(\mathcal{K}, t, \mathcal{W}) \models \psi$.

For $i \in [0, \kappa + 1]$, let $F_0 := S$ and $F_{i+1} := \|\psi\|^\mathcal{K}_{\mathcal{V}[X:=F_i]}$. Since $\kappa = |S| + 1$, it follows by Knaster-Tarski that $\|\varphi\|^\mathcal{K}_\mathcal{V} = F_{\kappa+1}$. In particular, $(\mathcal{K}, s, \mathcal{V}) \models \varphi$ iff $(\mathcal{K}, s, \mathcal{V}[X := F_\kappa]) \models \psi$. Let $\mathcal{G}^0 \dots \mathcal{G}^\kappa$ be the unfolding of $\mathcal{G}$. It is straightforward to check by induction on i that $\mathcal{G}^i$ is identical to $\mathcal{G}^-_{\mathcal{V}[X:=F_i]}$ for each $i \in [0, \kappa]$. By Lemma 5, we therefore obtain that Player 0 wins $\mathcal{G}$ from $((s, \varphi), E \dots E)$ iff he wins $\mathcal{G}^\kappa$ from $((s, \varphi), E^- \dots E^-)$. We can now conclude the case of greatest fixed-points: Player 0 wins $\mathcal{G}$ from $((s, \text{init}(\varphi)), E \dots E)$ iff he wins $\mathcal{G}$ from $((s, \varphi), E \dots E)$ iff he wins $\mathcal{G}^\kappa$ from $((s, \varphi), E^- \dots E^-)$ iff he wins $\mathcal{G}^-_{\mathcal{V}[X:=F_\kappa]}$ from $((s, \varphi), E^- \dots E^-)$ iff $(\mathcal{K}, s, \mathcal{V}[X := F_\kappa]) \models \psi$ iff $(\mathcal{K}, s, \mathcal{V}) \models \varphi$.

Note that the cases $\varphi = \varphi_1 \wedge \varphi_2$, $\varphi = \Box_a \psi$, $\varphi = \boxdot_a \psi$, and $\varphi = \mu X.\psi$ are dual to the cases $\varphi = \varphi_1 \vee \varphi_2$, $\varphi = \Diamond_a \psi$, $\varphi = \diamonddot_a \psi$, and $\varphi = \nu X.\psi$. ☐

5 Conclusion

We augmented the μ-calculus with a transition-deleting modality, which yields the fixed-point logic SL_μ over dynamically changing structures. We have seen that model checking is not algorithmically harder than model checking the sabotage modal logic without fixed-points. We introduced backup games as extended parity games with the feature of edge deletion and of storing and restoring the current arena in a fixed number of registers. Even when the access to registers has to follow a stack discipline, these games serve as model checking games for SL_μ. The problem of solving these games is PSPACE-complete. The games without limited access become EXPTIME-complete.

Model checking for SL_μ via backup games is not optimal yet: We could only show that the problem of solving backup games belongs to PSPACE for a fixed number of registers. But the number of registers of the game $\mathcal{G}_{\mathcal{K},\varphi,\mathcal{V}}$ is equal to the fixed-point depth of φ. Our result yields, so far, only a PSPACE-procedure for formulae with bounded fixed-point depth. Note that we already know that the model checking with a fixed formula can be done in polynomial time with respect to the size of the structure, cf. Lemma 1. It remains an open question

whether backup games can be solved in polynomial space, regardless of the number of registers. If we can answer this question positively, then we would obtain an optimal model checking procedure for SL_μ. Otherwise, there may be a subclass of games that serve as model checking games, but which can be solved in polynomial space regardless of the number of registers. Finally, it remains open whether we can express the winning condition for backup games as SL_μ-formulae. Recall that this is the case for standard parity games and L_μ [14]. If such a translation yields SL_μ-formulae with a size polynomial in the size of the arena, then this would also give a positive answer to the above question.

Note that SL_μ does not provide a way to express, for example, a general reachability *while* transitions are deleted (due to the restoration in inductive fixed-point constructions). It is worth to study logics that allow such an overlap of fixed-points and deletion. Finally, sabotage logics that allow to address the deletion of objects are only a first step towards a general theory of logics over dynamically changing structures.

References

1. Balcázar, J.L., Díaz, J., Gabarró, J.: Structural Complexity II. Springer 1990
2. van Benthem, J.: An essay on sabotage and obstruction. In: Mechanizing Mathematical Reasoning. LNAI 2605 (2005), 268–276
3. Bernholtz, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. In: CAV '94. LNCS 818 (1994), 142–155
4. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: FOCS '91 (1991), 368–377
5. Emerson, E.A., Jutla, C.S., Sistla, A.P.: On model-checking for fragments of μ-calculus. In: CAV '93. LNCS 697 (1993), 385–396
6. Jurdziński, M.: Small progress measures for solving parity games. In: STACS '00. LNCS 1770 (2000), 290–301
7. Kozen, D.: Results on the propositional μ-calculus. TCS **27** (1983), 333–354
8. Löding, Ch., Rohde, Ph.: Solving the sabotage game is PSPACE-hard. In: MFCS '03. LNCS 2747 (2003), 531–540
9. Löding, Ch., Rohde, Ph.: Model checking and satisfiability for sabotage modal logic. In: FSTTCS '03. LNCS 2914 (2003), 302–313
10. Rohde, Ph.: On Games and Logics over Dynamically Changing Structures. Technical report submitted as dissertation thesis at RWTH Aachen (2005). Available under `www-i7.informatik.rwth-aachen.de/~rohde/thesis.pdf`
11. Stockmeyer, L.J., Chandra, A.K.: Provably difficult combinatorial games. SIAM Journal on Computing **8** (1979), 151–174
12. Vardi, M.Y.: The complexity of relational query languages. In: STOC '83 (1982), 137–146
13. Vardi, M.Y.: On the complexity of bounded-variable queries. In: PODS '95 (1995), 266–276
14. Walukiewicz, I.: Monadic second-order logic on tree-like structures. TCS **275** (2002), 311–346

A Finite Model Construction for Coalgebraic Modal Logic

Lutz Schröder

Department of Computer Science, University of Bremen

Abstract. In recent years, a tight connection has emerged between modal logic on the one hand and coalgebras, understood as generic transition systems, on the other hand. Here, we prove that (finitary) coalgebraic modal logic has the finite model property. This fact not only reproves known completeness results for coalgebraic modal logic, which we push further by establishing that every coalgebraic modal logic admits a complete axiomatization of rank 1; it also enables us to establish a generic decidability result and a first complexity bound. Examples covered by these general results include, besides standard Hennessy-Milner logic, graded modal logic and probabilistic modal logic.

1 Introduction

Coalgebra has recently had increasing success as a generic theory of reactive systems, providing a unifying perspective on a wide variety of system types [21]. Many concepts of concurrency theory can be cast in the coalgebraic framework; this includes general notions of bisimulation, coinduction, and corecursion, as well as a generic modal logic [10, 19, 12, 15, 17]. The role of this *coalgebraic modal logic* is twofold: on the one hand, one obtains a suitable generic reactive specification language, which respects encapsulation of the state space, i.e. relates well to behavioral equivalence of states [17, 23], and is sufficiently intuitive for use in actual software specification languages, including object-oriented specification [20, 12, 14]. On the other hand, coalgebraic modal logics frequently correspond to known modal logics such as graded modal logic or probabilistic modal logic, and thus provide these logics with a coalgebraic semantics.

In [16] and subsequent work [5, 11], a (necessarily weak) completeness result for coalgebraic modal logic has been established stating that a deductive system consisting of propositional entailment, a congruence rule, and a given set of axioms of rank 1 is weakly complete, provided that the axioms are in a precise sense sufficiently strong; the latter property is referred to as *reflexivity*. Here, we exhibit a finite model construction which relies on reflexivity. We thus reprove the mentioned weak completeness result. Moreover, we show that every coalgebraic modal logic admits a reflexive axiomatization, which then implies that coalgebraic modal logic has the *finite model property*, i.e. every satisfiable formula is satisfiable in a finite model. We further exploit the finite model construction to obtain a generic decision procedure which reduces the satisfiability

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 157–171, 2006.
© Springer-Verlag Berlin Heidelberg 2006

problem for a coalgebraic modal logic to the much simpler *one-step satisfiability* problem. This yields not only decidability of a large number of modal logics, including the above-mentioned graded and probabilistic modal logics, but also, under mild conditions, a first complexity bound.

The material is organized as follows. Section 2 gives an introduction to coalgebra and coalgebraic modal logic, including a number of examples. In Section 3, we recall the deduction system of coalgebraic modal logic [16, 5, 11] and the above-mentioned notion of reflexivity, and prove that reflexive axiomatizations always exist. We then prove the finite model property in Section 4, from which we obtain the generic decision procedure and the arising complexity bound in Section 5.

2 Coalgebraic Modal Logic

We briefly recapitulate the basics of the coalgebraic modelling of reactive systems and of the specification of such systems by means of coalgebraic modal logic.

Definition 1. Let $T : \mathbf{Set} \to \mathbf{Set}$ be a functor (in this work, all functors will implicitly be set functors), referred to as the *signature functor*. A *T-coalgebra* $A = (X, \xi)$ consists of a set X of *states* and an *evolution map* $\xi : X \to TX$. A *morphism* $(X_1, \xi_1) \to (X_2, \xi_2)$ of T-coalgebras is a map $f : X_1 \to X_2$ such that $\xi_2 \circ f = Tf \circ \xi_1$. A T-coalgebra C is called *final* if there exists, for each T-coalgebra A, a unique morphism $A \to C$.

Intuitively, the evolution map describes the successor states of a state, organized in a data structure given by T. These data encode the observable *behavior* of a system, and morphisms of coalgebras preserve this behavior.

We shall occasionally require a technical condition on the signature functor:

Definition 2. A functor T is called *κ-accessible*, where κ is a regular cardinal, if T preserves κ-directed colimits.

Intuitively, this amounts to a restriction on the branching degree, stating that every state in a T-coalgebra has less than κ successors. Many of the examples given below are ω-accessible; however, the central results presented here hold for arbitrary functors.

We explicitly fix some logical terminology:

Definition 3. Let T be a functor. A *language for T-coalgebras* is a set $\mathcal{L}$ of *formulae*, equipped with a family of *satisfaction* relations $\models_C$ (or just $\models$) between states of T-coalgebras $C = (X, \xi)$ and formulae $\phi \in \mathcal{L}$; we define $[\![\phi]\!]_C$ (or just $[\![\phi]\!]$) as the set $\{x \in X \mid x \models_C \phi\}$. The coalgebra C *satisfies* a formula ϕ if $x \models \phi$ for all $x \in X$; in this case, we write $C \models \phi$. We say that C *satisfies* a set Φ of formulae, and write $C \models \Phi$, if $C \models \phi$ for each $\phi \in \Phi$.

A formula $\psi \in \mathcal{L}$ is a *global consequence* of a set $\Phi \subset \mathcal{L}$ if, for every T-coalgebra C, $C \models \psi$ whenever $C \models \Phi$. In this case, we write $\Phi \models_g \psi$. We say that ψ is *valid* if $\emptyset \models_g \psi$. Moreover, ψ is a *local consequence* of Φ if, for every

state x in every T-coalgebra, $x \models \psi$ whenever $x \models \Phi$. A set Φ of formulae is *locally satisfiable* if it is satisfied in some state in some T-coalgebra; Φ is *globally satisfiable* if $\Phi \not\models_g \bot$ (i.e. if Φ is satisfied by some non-empty T-coalgebra).

Every local consequence is also a global consequence, and every globally satisfiable formula is locally satisfiable. A formula ϕ is valid iff $\neg\phi$ is locally unsatisfiable.

As a specification logic for coalgebraically modelled reactive systems, *coalgebraic modal logic* in the form considered here has been introduced in [17], generalizing previous results [10, 19, 12, 15]. The semantics is based on the following central notion.

Definition 4. A *predicate lifting* for a functor T is a natural transformation

$$\lambda : 2^{-} \to 2^{T},$$

where 2^{-} denotes the contravariant powerset functor $\mathbf{Set}^{op} \to \mathbf{Set}$, with $2^{f}(A) = f^{-1}[A]$.

A predicate lifting λ induces a *transposite* $\lambda^{\flat} : T \to 2^{(2^{-})}$, given by $\lambda^{\flat}_X(t) = \{A \subset X \mid t \in \lambda_X(A)\}$. A set Λ of predicate liftings for T is called *separating* if for each set X, the source of maps $(\lambda^{\flat}_X : T \to 2^{(2^{-})})_{\lambda \in \Lambda}$ is jointly injective.

In the terminology introduced above, a coalgebraic modal logic is a language $\mathcal{L}^{\kappa}(\Lambda)$ for T-coalgebras, determined by a set Λ of predicate liftings for T and a regular cardinal κ which serves as a bound for conjunctions. Since we are aiming at finite model results here, we restrict the exposition to the finite case $\kappa = \omega$, and we write $\mathcal{L}(\Lambda)$ in place of $\mathcal{L}^{\omega}(\Lambda)$. Formulae $\phi, \psi \in \mathcal{L}(\Lambda)$ are defined by the grammar

$$\phi ::= \bot \mid \phi \wedge \psi \mid \neg\phi \mid [\lambda]\,\phi,$$

where λ ranges over Λ. Disjunctions $\phi \vee \psi$, truth $\top$, and other boolean operations are then defined as usual. In the definition of satisfaction, the clauses for boolean operators are as expected; the clause for the modal operator $[\lambda]$ is

$$x \models_{(X,\xi)} [\lambda]\,\phi \iff \xi(x) \in \lambda_X [\![\phi]\!]_{(X,\xi)}.$$

The *size* $|\phi|$ of a formula ϕ is the number of subformulae of ϕ.

Remark 5. It is shown in [17, 23] that if Λ is separating and T is ω-accessible, then $\mathcal{L}(\Lambda)$ is *adequate* and *expressive*: states x and y in T-coalgebras A and B, respectively, satisfy the same $\mathcal{L}(\Lambda)$-formulae iff they are *behaviorally equivalent* in the sense that there exists a coalgebra C and morphisms $f : A \to C$, $g : B \to C$ such that $f(x) = g(y)$. This is one reason why modal logic is regarded as a suitable means of expression for coalgebraic specification — it automatically ensures encapsulation of the state space, allowing judgements precisely about the observable behavior of states.

Remark 6. By the results of [23], obtaining an expressive logic for a given ω-accessible functor may require the use of polyadic modal operators obtained from

polyadic predicate liftings. The results of this paper extend straightforwardly to polyadic modal logic; we restrict the exposition to the unary case purely in the interest of readibility.

Example 7. [17, 5, 23]

1. Let $\mathcal{P}$ be the covariant powerset functor. Then $\mathcal{P}$-coalgebras are graphs, thought of as transition systems or indeed Kripke frames. A separating set of predicate liftings is formed by the single predicate lifting $\lambda^\forall$ defined by

$$\lambda^\forall_X(A) = \{B \subset X \mid B \subset A\}.$$

   This lifting gives rise to the standard box modality $\Box = [\lambda^\forall]$. All this is easily adapted to transition systems with branching degree limited by a regular cardinal κ, described as coalgebras for the κ-accessible functor $\mathcal{P}_\kappa$ defined by $\mathcal{P}_\kappa(X) = \{A \subset X \mid |A| < \kappa\}$.
2. It is straightforward to extend a given coalgebraic modal logic for T with a set V of propositional symbols. This amounts to considering the functor $T \times \mathcal{P}(V)$, where $\mathcal{P}(V)$ stands for the corresponding constant functor. Separation is then ensured by adding predicate liftings λ^a, $a \in V$, defined by

$$\lambda^a_X(A) = \{(t, B) \in TX \times \mathcal{P}(V) \mid a \in B\}.$$

   Since λ^a is independent of its argument, the induced modal 'operator' can be written as just the propositional symbol a, with the expected meaning.
3. The *finite multiset* (or *bag*) functor $\mathcal{B}_\mathbb{N}$ is given as follows. The set $\mathcal{B}_\mathbb{N}(X)$ consists of the maps $B : X \to \mathbb{N}$ with finite support; we say that B contains $x \in X$ with *multiplicity* $B(x)$. We write multisets additively, denoting by $\sum n_i x_i$ the multiset that contains x with multiplicity $\sum_{x_j = x} n_j$. For $f : X \to Y$, $\mathcal{B}_\mathbb{N}(f)(\sum n_i x_i) = \sum n_i f(x_i)$. Coalgebras for $\mathcal{B}_\mathbb{N}$ are directed graphs with $\mathbb{N}$-weighted edges, often referred to as *multigraphs* [6].

   A separating set of predicate liftings λ^k, $k \in \mathbb{N}$, is defined by

$$\lambda^k_X(A) = \{\textstyle\sum n_i x_i \in \mathcal{B}_\mathbb{N} X \mid \sum_{x_i \in A} n_i > k\}.$$

   The arising modal operators are exactly the modalities $\Diamond_k$ of *graded modal logic* (cf. e.g. [6]), i.e. $x \models \Diamond_k \phi$ iff ϕ holds for more than k successor states of x, taking into account multiplicities. Note that $\Box_k$, defined as $\neg\Diamond_k\neg$, is monotone, but fails to be normal unless $k = 0$. (Recall that a modal operator $\Box$ is called *monotone* if it satisfies $\Box(p \wedge q) \to \Box p$, and *normal* if it satisfies $\Box(p \to q) \to \Box p \to \Box q$).
4. A similar functor, denoted $\mathcal{B}_\mathbb{Z}$, is given by a slight modification of the multiset functor where we allow elements to have also *negative* multiplicities, i.e. $\mathcal{B}_\mathbb{Z} X$ consists of finite maps $X \to \mathbb{Z}$, called *generalized multisets* (this set is also familiar as the free abelian group over X).

   A separating set of predicate liftings λ^k, $k \in \mathbb{Z}$, with induced modal operators $\Diamond_k$ is defined analogously as for multisets. Note that $\Box_k$ fails to be monotone even for $k = 0$. One may imagine an interpretation of $\mathcal{B}_\mathbb{Z}$-coalgebras

as transition systems that allow some form of trading — formulae may be violated in 'positive' successor states, as long as this is counterbalanced by violations in 'negative' successor states.

5. The *finite distribution functor* D_ω maps a set X to the set of probability distributions on X with finite support. Coalgebras for the functor $T = D_\omega \times \mathcal{P}(V)$, where $V \neq \emptyset$ is a set of propositional symbols, are probabilistic transition systems (also called *probabilistic type spaces* [8]) with finite branching degree. (The example is easily extended to countable branching by considering instead the functor D_{ω_1} of probability distributions with countable support, while higher branching degrees require a more elaborate measure theoretic treatment [27].)

   A separating set for T is obtained by combining the propositional symbols (cf. Example 2) with the predicate liftings λ^p, $p \in [0,1] \cap \mathbb{Q}$, defined by

$$\lambda^p(A) = \{P \in D_\omega X \mid PA \geq p\}.$$

   These induce the modal operators $\langle p \rangle = [\lambda^p]$ of *probabilistic modal logic (PML)* [13, 8], where $\langle p \rangle\, \phi$ reads 'ϕ holds in the next step with probability at least p'. Note that $[p]$, defined as $\neg\langle p \rangle\neg$, is monotone, but not normal.

6. For a field k, the *linear space functor* $k \cdot _$ takes a set X to the free k-vector space $k \cdot X$, i.e. the set of formal k-linear combinations, over X. A coalgebra for $k \cdot _$ is a *linear automaton* [3, 26] (where one would in general also assume linear output in a vector space V, corresponding to the functor $(k \cdot _) \times V$). In the case $k = \mathbb{R}$, a separating set of predicate liftings can be constructed in the same way as for D_ω, giving rise to modal operators $\langle p \rangle$ for $p \in \mathbb{Q}$. Here, $\langle p \rangle \phi$ holds if the sum of the coefficients of successor states satisfying ϕ is at least p.

7. The above examples may be extended by adding inputs from an alphabet I, i.e. by passing from T to one of the functors S and R given by $SX = I \to TX$ and $RX = T(I \times X)$, respectively. When I is finite, these functors are isomorphic for $T \in \{\mathcal{P}_\omega, \mathcal{B}_\mathbb{N}, \mathcal{B}_\mathbb{Z}\}$ but not for $T = D_\omega$. In the latter case, S-coalgebras are reactive probabilistic automata, and R-coalgebras are generative probabilistic automata [1] (more precisely, one allows for terminal states by additionally introducing the constant functor 1 as a summand).

   An expressive set of modal operators is then obtained by indexing modal operators over $a \in I$ in the form $[_]_a$. In the case $T = \mathcal{P}_\omega$, this leads to the usual operators $\boxed{a}$ of Hennessy-Milner Logic [9]. In the probabilistic case, the meaning of $\langle p \rangle_a\, \phi$ in reactive probabilistic automata is that on input a, ϕ holds in the next step with probability at least p, and in generative probabilistic automata that with probability at least p, the input is a and ϕ holds in the next step.

Remark 8. Graded modal logic is more standardly interpreted over Kripke models by just counting successor states (as e.g. in [25]), rather than in multigraphs as in the above example and e.g. in [6]. One can regard Kripke models as multigraph models by just regarding sets as multisets where all elements have

multiplicity 1; conversely, one can unroll a state in a multigraph model into a tree-like Kripke model by making copies of elements according to their multiplicity. Both constructions preserve satisfaction of graded modal formulae, so that the two semantics induce the same local consequence relations.

3 Proof Systems for Coalgebraic Modal Logic

We now discuss completeness of derivation for coalgebraic modal logic, partly following [16, 5, 11]. Since an otherwise unstructured signature functor T contains information only about the one-step evolution of the system (as opposed to a comonad, which may contain information also about further steps), it is natural to expect that for the axiomatization of a coalgebraic modal logic for T it is enough to consider modal axioms of rank 1. The approach taken in [16, 5, 11] is based on this expectation; we shall prove below that it is indeed formally the case that axioms of rank 1 are sufficient. This fact will be crucial for our finite model result to be proved in Section 4.

To begin, we note that both the global and the local consequence relation (Definition 3) of a coalgebraic modal logic in general fail to be compact:

Example 9. In the case of Hennessy-Milner logic over finitely branching systems [9] with two inputs a, b, the set

$$\Phi = \{\langle a \rangle ([b]^{n+1}\bot \wedge \langle b \rangle^{n}\top) \mid n \in \mathbb{N}\},$$

where $[b]^n$ stands for n consecutive boxes, is locally (and hence globally) unsatisfiable, since it requires, for each n, the existence of an a-successor from which exactly n b-steps are possible. However, every finite subset of Φ is globally (and hence locally) satisfiable. For an example of the same kind, but of bounded rank, consider the set

$$\{\Diamond_k \top \mid k \in \mathbb{N}\}$$

of graded modal formulae over $\mathcal{B}_{\mathbb{N}}$. Non-compactness of PML is observed in [8]; in this case, non-compactness does *not* have to do with bounded branching.

Thus, in general neither the local nor the global consequence relations of a coalgebraic modal logic admit a finitary complete proof system. Instead, one is lead to study weak completeness, where a proof system is called *weakly complete* if it proves all valid formulae. This notion is equivalent to completeness in the sense used in [5, 11, 16], where only local consequence with singleton sets of premises is considered (ψ is a local consequence of $\{\phi\}$ iff $\phi \to \psi$ is valid).

For the remainder of the paper, we assume given a functor T and a set Λ of predicate liftings for T. We recall a few basic notions from propositional logic, as well as notation for coalgebraic modal logic introduced in [16, 5]:

Definition 10. Let V be a set. We denote the set of propositional formulae over V by $\mathsf{Prop}(V)$. Here, we regard $\neg$ and $\wedge$ as the basic connectives, with all other connectives defined in the standard way. A *literal* over V is either an

element of V or the negation of such an element. A *(conjunctive) clause* is a finite, possibly empty, disjunction (conjunction) of literals. Moreover, we denote by $\mathsf{Up}(V)$ the set $\{[\lambda]a \mid \lambda \in \Lambda, a \in V\}$.

If the elements of V are, or have an interpretation as, subsets of a given set X, then $\phi \in \mathsf{Prop}(V)$ can be interpreted as a subset $[\![\phi]\!]_X$ of X; we say that ϕ *holds in* X and write $\models_X \phi$ if $[\![\phi]\!]_X = X$, and we say that ϕ is *satisfiable in* X if $[\![\phi]\!]_X \neq \emptyset$. Similarly, if $a \in V$ is interpreted as a subset A of X, then we interpret $[\lambda]a \in \mathsf{Up}(V)$ as the subset $[\![[\lambda]a]\!] = \lambda_X(A)$ of TX. (This can of course be iterated, leading to interpretations $[\![\phi]\!] \subset TX$ of $\phi \in \mathsf{Up}(\mathsf{Prop}(V))$ etc.)

In case the elements of V are formulae in $\mathcal{L}(\Lambda)$, we also regard propositional formulae over V as formulae in $\mathcal{L}(\Lambda)$. We sometimes explicitly designate V as consisting of *propositional variables*; propositional variables retain their status across further applications of Up and Prop (e.g. if V is a set of propositional variables, then V and not $\mathsf{Prop}(V)$ is the set of propositional variables for $\mathsf{Up}(\mathsf{Prop}(V))$). Given a set L, an L*-substitution* is a substitution σ of the propositional variables by elements of L. Then, $\phi\sigma$ is called an *instance* of ϕ over L. If $L \subset \mathcal{P}(X)$ for some X, then we also refer to σ as an L*-valuation* or a $\mathcal{P}(X)$*-valuation*.

The format we impose on axioms is essentially equivalent to the formal notion of *axiom* used in [16, 11]:

Definition 11. A *rank-*1*-clause* over a set V of propositional variables is a clause over $\mathsf{Up}(\mathsf{Prop}(V))$. Such a clause is *valid* if all its instances over $\mathcal{L}(\Lambda)$ are valid.

Proposition 12. *Let ϕ be a rank-*1*-clause. If $\phi\sigma$ holds in TX for every set X and every $\mathcal{P}(X)$-valuation σ, then ϕ is valid. The converse holds if T is ω-accessible, Λ is separating, and the final T-coalgebra is infinite.*

(The condition on ϕ in the above proposition has been called *admissibility* in [16], where also the first implication is proved.)

A given set Ax of valid rank-1-clauses, called *axioms*, induces a proof system for $\mathcal{L}(\Lambda)$ as follows [11].

Definition 13. The set of formulae *derivable* from Ax is the smallest set containing all instances of axioms over $\mathcal{L}(\Lambda)$ and closed under propositional entailment and the *congruence rule*

$$\frac{\phi \leftrightarrow \psi}{[\lambda]\phi \leftrightarrow [\lambda]\psi}.$$

It is easy to see that this proof system is sound. The completeness results in [16, 5, 11] require the presence of 'enough' axioms in the following sense.

Definition 14. The set Ax is *reflexive* if, for each set X and each $\mathfrak{A} \subset \mathcal{P}(X)$, every clause ϕ over $\mathsf{Up}(\mathfrak{A})$ that holds in TX is *derivable*, i.e. propositionally entailed by instances of axioms over $\mathfrak{A}$ and by formulae of the form $[\lambda]\phi \leftrightarrow [\lambda]\psi$, where $\phi, \psi \in \mathsf{Prop}(\mathfrak{A})$ and $\phi \leftrightarrow \psi$ holds in X.

Of course, we can restrict ourselves to finite $\mathfrak{A}$. The definition originally used in [16] to establish weak completeness is stronger than the notion above in that derivations of clauses over L are restricted to use only the subset relation on $\mathfrak{A}$ rather than propositional formulae that hold in X. By the results of [5], the weaker definition above, which is essentially equivalent to one given in [11], suffices to establish weak completeness. Examples of reflexive axiomatizations are given in [16, 5].

The weak completeness theorem, stating that reflexive sets induce weakly complete proof systems [16, 5], will appear as a corollary to our finite model result in Section 4. We now proceed to establish that every functor indeed admits a reflexive set, i.e. that the set of all valid rank-1-clauses is reflexive.

As a preparation, we note that rank-1-clauses are equivalent to rules of a restricted format.

Definition 15. An *(extended) one-step rule* R over a set V of propositional variables is a rule ϕ/ψ, where $\phi \in \mathsf{Prop}(V)$, and ψ is a clause over $\mathsf{Up}(V)$ (over $\mathsf{Up}(\mathsf{Prop}(V))$). The rule R is *valid* if, whenever $\phi\sigma$ is valid for an $\mathcal{L}(\Lambda)$-substitution σ, then $\psi\sigma$ is valid. As part of a proof system, R allows deriving $\psi\sigma$ from $\phi\sigma$ for each $\mathcal{L}(\Lambda)$-substitution σ.

Thanks to the congruence rule, one-step rules can replace extended one-step rules (just introduce premises abbreviating propositional formulae as propositional variables). In particular, every rank-1-clause can be replaced by a one-step rule. Conversely, we have

Proposition 16. *For each one-step rule R over V, there exists a rank-1-clause χ over V such that χ and R are derivable from each other by propositional reasoning and the congruence rule.*

The proof needs the following fact from propositional logic.

Lemma 17. *Let $\phi \in \mathsf{Prop}(V)$ be satisfiable. Then there exists a $\mathsf{Prop}(V)$-substitution σ such that*

$$\begin{aligned} &\phi \to (a \leftrightarrow \sigma(a)) \quad \textit{(for each } a \in V\textit{) and} \\ &\phi\sigma \end{aligned}$$

are tautologies.

Proof (Proposition 16). We can assume that the the premise ϕ of $R = \phi/\psi$ is satisfiable. Thus, fix σ as in the above lemma for ϕ. Then R and the rank-1-clause $\psi\sigma$ are mutually interderivable as claimed. □

We are now ready to prove the announced axiomatizability result:

Theorem 18. *The set of all valid rank-1-clauses is reflexive.*

Proof. (Sketch) Let X be a set, let $\mathfrak{A} \subset \mathcal{P}(X)$ be finite, and let the clause ψ over $\mathsf{Up}(\mathfrak{A})$ hold in TX. Let the formula ϕ be the 'propositional theory' of $\mathfrak{A}$, i.e. the (finite) conjunction of all clauses over $\mathfrak{A}$ that hold in X. Then one can

show that the one-step rule $R \equiv \phi/\psi$ over $\mathfrak{A}$, abused as a set of propositional variables, is valid. By Proposition 16, it follows that R is derivable from the set of all valid rank-1-clauses; combined with the fact that ϕ holds in X, this yields a derivation of ψ over TX in the sense of Definition 14. $\square$

4 The Finite Model Construction

The non-compactness of coalgebraic modal logic (cf. Example 9) means that canonical models, based on the set of all maximally consistent sets w.r.t. a finitary deduction system, do not in general exist. An alternative is to use filtration methods (cf. e.g. [4, 2]), in the variant that uses consistent subsets of closed sublanguages.

We recall a few basic definitions:

Definition 19. Given a set Ax of axioms, a finite set $\{\phi_1, \dots, \phi_n\}$ of formulae is called *consistent* (w.r.t Ax) if $\neg(\phi_1 \wedge \dots \wedge \phi_n)$ is not derivable according to Definition 13. A set Σ of formulae is called *closed* if it is closed under subformulae and under *normalized negation* $\sim$, where $\sim\phi$ is defined to be ψ in case ϕ is of the form $\neg\psi$, and $\neg\phi$ otherwise. A subset A of Σ is called a *Σ-Hintikka set* if $\bot \notin A$ and, for $\phi, \psi \in \Sigma$, $\phi, \psi \in A$ iff $\phi \wedge \psi \in A$, and, for $\neg\phi \in \Sigma$, $\neg\phi \in A$ iff $\phi \notin A$. Moreover, A is called a *Σ-atom* if A is maximal among the consistent subsets of Σ.

Thus, a Σ-atom is just a consistent Σ-Hintikka set.

Lemma 20 (Lindenbaum Lemma). *Every consistent subset of Σ is contained in a Σ-atom.*

Given a closed set Σ, the carrier of the model to be constructed will be the set S of Σ-atoms; the main problem is then to define the coalgebra structure on S. The following lemma is crucial for this purpose.

Lemma 21. *Let V be a set of propositional variables, let $\phi \in \mathsf{Prop}(V)$, and let σ be a Σ-substitution. Then $\phi\sigma$ is derivable iff $\phi\tau$ holds in the set S of Σ-atoms, where τ is the $\mathcal{P}(S)$-valuation given by $\tau(a) = \{A \in S \mid \sigma(a) \in A\}$.*

Expecting that the extension of a formula $\phi \in \Sigma$ in the coalgebra (S, ξ) to be constructed will be the set $\{A \in S \mid \phi \in A\}$, we will need to require that

$$\xi(A) \in \lambda_S\{B \in S \mid \phi \in B\} \iff [\lambda]\phi \in A \qquad (*)$$

for all $A \in S$ and all formulae $[\lambda]\phi$ in Σ. This is where reflexivity comes in:

Lemma 22 (Existence Lemma). *If Ax is reflexive and Σ is finite, then $\xi(A)$ satisfying $(*)$ exists for each $A \in S$.*

Proof. Assume that $\xi(A)$ does *not* exist. Let V be the set $\{a_\phi \mid \phi \in \Sigma\}$ of propositional variables. Let ψ be the clause over $\mathsf{Up}(V)$ containing, for each $[\lambda]\phi \in \Sigma$, the literal $\neg[\lambda]a_\phi$ if $[\lambda]\phi \in A$, and the literal $[\lambda]a_\phi$ otherwise. Let τ be the $\mathcal{P}(S)$-valuation taking a_ϕ to $\{B \mid \phi \in B\}$. Then $\psi\tau$ holds in TS by assumption. By

reflexivity, $\psi\tau$ is derivable in the sense of Definition 14 from the propositional formulae of the form $\chi\tau$ that hold in S. This derivation can be copied to obtain a derivation of $\psi\sigma$ from those $\chi\sigma$ for which $\chi\tau$ holds in S, where σ is the Σ-substitution taking a_ϕ to ϕ. These $\chi\sigma$ are derivable by Lemma 21. Thus, $\psi\sigma$ is derivable, in contradiction to the consistency of A. □

It remains to prove the truth lemma, which we state in a slightly more general form than needed in this section for reuse in Section 5:

Lemma 23 (Truth Lemma). *Let Σ be a closed set, let S be a set of Σ-Hintikka sets, and let $\xi : S \to TS$ satisfy condition $(*)$ above. Then for all $\phi \in \Sigma$ and all $A \in \Sigma$,*

$$A \models_{(S,\xi)} \phi \iff \phi \in A.$$

Proof. Straightforward induction over ϕ. □

This is all we need in order to establish

Theorem 24. *Let Ax be reflexive. Then every formula ϕ that is consistent w.r.t. Ax is locally satisfiable in a finite T-coalgebra of size at most $2^{|\phi|}$.*

The proof is just an application of the lemmas above to the (finite) smallest closed set $\Sigma(\phi)$ containing a given consistent formula ϕ.

As announced, the above result implies weak completeness [16, 5]; explicitly:

Corollary 25 (Weak completeness). *The proof system induced by a reflexive set of axioms is weakly complete.*

Combining Theorems 18 and 24, we obtain independently of deduction:

Corollary 26 (Finite model property). *Every locally satisfiable formula ϕ is locally satisfiable in a finite T-coalgebra of size at most $2^{|\phi|}$.*

Remark 27. The finite model property does not generalize to the case where global axioms are present, i.e. it may be the case that ϕ is locally satisfiable in some model globally satisfying a formula ψ, but not in any finite such model. Examples are found already in standard modal logic [18].

5 Decidability

Unlike in the classical case, the finite model construction of the preceding section does not immediately imply decidability, even though it gives a computable bound on the size of the model, since there may in general be infinitely many T-coalgebras on a given finite set. (In fact, this is the interesting case; for functors T that preserve finite sets, a finite model construction is given already in [16].) If T takes finite sets to recursively enumerable sets — as is the case e.g. for $\mathcal{B}_\mathbb{N}$, $\mathcal{B}_\mathbb{Z}$, and $\mathbb{Z}[_]$, but not for D_ω — then the finite model property implies that the set of satisfiable formulae is r.e.. We then obtain decidability provided that the set of valid formulae is also r.e., which by the weak completeness theorem and Theorem 18 is the case if the set of all valid rank-1-clauses is r.e.

We can however improve on this by exploiting the details of the finite model construction, as follows. We have no direct access to the set of all $\Sigma(\phi)$-atoms, since this would already require a decision procedure for consistency. We can however easily decide the set H of $\Sigma(\phi)$-Hintikka sets. We are then faced with the following decision problem:

Definition 28. The *one-step satisfiability* problem is to decide, given a finite set X and a *conjunctive* clause ϕ over $\mathsf{Up}(\mathcal{P}(X))$, whether ϕ is satisfiable in TX.

Remark 29. For purposes of determining the input size for the above problem, we assume that subsets of X can be represented in $|X|$ bits. Moreover, we assume that Λ is countable, with a reasonable size measure for the representation of modal operators supposed as given (e.g., size $\log k$ for the graded modal operator $[k]$, and size $\log n + \log m$ for the probabilistic modal operator $[n/m]$, with $n, m \in \mathbb{N}$ relatively prime). The same applies to the size of a formula ϕ as input for the satisfiability problem, which is thus larger than the size $|\phi|$ as defined in Section 2.

A decision procedure for one-step satisfiability leads to the following algorithm for satisfiability of ϕ.

Algorithm 1. For all subsets S of H, perform the following steps.

1. Check whether $\phi \in A$ for some $A \in S$; if not, continue with the next S.
2. Decide whether for all $A \in S$, the conjunctive clause

$$\bigwedge_{[\lambda]\psi \in A} [\lambda]\{B \in S \mid \psi \in B\} \wedge \bigwedge_{\neg[\lambda]\psi \in A} \neg[\lambda]\{B \in S \mid \psi \in B\}$$

   is satisfiable in TS. If yes, terminate with output 'yes'; otherwise, continue in Step 1 with the next S.

If all S have been checked unsuccessfully, terminate with output 'no'.

The correctness of the algorithm is guaranteed by the Truth Lemma. Thus,

Theorem 30. *If one-step satisfiability is decidable, then satisfiability of $\mathcal{L}(\Lambda)$-formulae is decidable.*

A non-deterministic variant of Algorithm 1 will also be useful:

Algorithm 2. Nondeterministically choose $S \subset H$; then proceed as in Algorithm 1, but fail (i.e. loop infinitely) rather than continue with the next S if one of the checks in Steps 1 or 2 fails.

In this algorithm, we can also employ a *semi*-decision procedure for one-step satisfiability. Since acceptance sets of non-deterministic algorithms are r.e., we thus have

Theorem 31. *If one-step satisfiability is semi-decidable, then satisfiability of $\mathcal{L}(\Lambda)$-formulae is semi-decidable.*

(Note that semi-decidability of one-step satisfiability is weaker than the above-mentioned condition that T takes finite sets to r.e. sets. E.g., the one-step satisfiability problem will turn out to be decidable for Λ as in Example 7.5, although $D_\omega(X)$ is uncountable for $|X| \geq 2$.)

Algorithm 2 yields the not overly tight complexity bound to be expected for filtration-based algorithms:

Theorem 32. *If the one-step satisfiability problem is in NP, then satisfiability of $\mathcal{L}(\Lambda)$-formulae is in NEXPTIME.*

Remark 33. In [5], logics for coalgebras are constructed in a modular fashion, following the structure of the signature functor; this raises the question of whether the above decidability and complexity results behave well w.r.t. these constructions. It is easy to see that decision procedures for one-step satisfiability can be combined along products and sums of functors and their logics, while this is not so clear for the case of functor composition $S \circ T$: here, one has to do with conjunctive clauses over $\mathsf{Up}(S(TX))$, where the application of T may produce an exponential blowup or lead to infinite sets.

Besides the examples whose decidability is already captured by the finite model result of [16], i.e. functors preserving finite sets, such as $\mathcal{P}$, our results cover the following cases.

Example 34.
1. Let Λ be the set of predicate liftings λ_k for the multiset functor of Example 7.3. Then the one-step satisfiability problem amounts to deciding the solvability of systems of linear inequations over the naturals; this problem is in *NP* [22]. By Theorem 32, we obtain that *the satisfiability problem for graded modal logic is in NEXPTIME*. (In fact, this problem is in *PSPACE* [25].)
2. By the same line of reasoning, the satisfiability problem for generalized graded modal logic over coalgebras for the generalized multiset functor (Example 7.4) is in *NEXPTIME*.
3. Let Λ be the set of predicate liftings for PML as in Example 7.5. Then the one-step satisfiability problem amounts to the solvability of systems of rational linear inequations over the reals, which is decidable in polynomial time by standard linear programming methods (using Motzkin's transposition theorem to treat also strict inequalities) [22]. By Theorem 30, it follows that *probabilistic modal logic is in NEXPTIME*.
4. By the same reasoning, the modal logic for linear automata of Example 7.6 is decidable in *NEXPTIME*
5. It is straightforward to extend the above results to include proposition symbols, where not already present, or inputs (cf. Examples 7.2 and 7.7).

Remark 35. A decision algorithm for PML is announced in [7], but not explicitly contained in the full version [8]. The latter proves the finite model property for PML, from which decidability does follow by the argument sketched at the beginning of this section, with some additional work required to reduce to models

with rational probabilities in order to ensure recursive enumerability of the set of finite models. Our algorithm improves this result by giving an upper bound on the complexity, albeit a rather generous one (see below).

Remark 36. It is, of course, desirable to obtain better general complexity bounds; this is the subject of ongoing research. The best general bound we can hope for is *PSPACE*, since the decision problem for K, i.e. the modal logic of $\mathcal{P}$, is known to be *PSPACE*-complete [2]. The following further results are forthcoming [24]:

(i) One can show by means of elimination of Hintikka sets (in the same manner as in known algorithms for PDL [2]) that satisfiability of $\mathcal{L}(\Lambda)$-formulae is in *EXPTIME* if one-step satisfiability is in P.
(ii) Given a tractable axiomatization of $\mathcal{L}(\Lambda)$, one can show that satisfiability of $\mathcal{L}(\Lambda)$-formulae is in *PSPACE* by means of a shallow model construction.

(Neither of these results makes Theorem 32 obsolete, since both rely on stronger assumptions.) By (i), one immediately improves the bound for PML as well as for the modal logic of linear automata from *NEXPTIME* (Example 34) to *EXPTIME*. The method of (ii) reproduces the known *PSPACE* bounds for K and for graded modal logic, and very likely leads to novel *PSPACE* bounds for all other logics mentioned in Example 34, including PML. An open problem that remains is whether there is a semantic criterion (not involving deduction) that guarantees a *PSPACE* bound.

We conclude this section with a few remarks on axiomatizability.

Definition 37. The *one-step validity* problem is to decide, given a finite set X and a (disjunctive) clause ϕ over $\mathsf{Up}(\mathcal{P}(X))$, whether ϕ holds in TX.

Of course, one-step validity and one-step satisfiability are, via negation, reducible to each other's complements.

Proposition 38. *Let T be ω-accessible, let Λ be separating, and let the final T-coalgebra be infinite. Then a rank-1-clause ϕ over a set V of propositional variables is valid iff $\phi\sigma$ holds in $T(\mathcal{P}(V))$, where σ is the $\mathcal{P}(\mathcal{P}(V))$-valuation taking $a \in V$ to the set*

$$\{B \in \mathcal{P}(V) \mid a \in B\}.$$

Corollary 39. *Let T be ω-accessible, let Λ be separating, and let the final T-coalgebra be infinite. Then the validity of rank-1-clauses is decidable (semi-decidable) if the one-step validity problem is decidable (semi-decidable).*

We have seen in Example 34 that the one-step validity problem is decidable in many important cases. Thus, Theorem 18 does frequently supply a feasible axiomatization of $\mathcal{L}(\Lambda)$, although one will, of course, in general strive for a more compact, preferably finite axiomatization.

6 Conclusion

Coalgebraic modal logic in general fails to be compact, so that completeness results are necessarily restricted to weak completeness and moreover cannot rely on constructing full canonical models. Above, we have described a finite model construction for coalgebraic modal logic, using the 'small canonical model' method known from standard modal logic; weak completeness is a corollary to this result. Here, the notion of reflexive axiom sets, which has appeared as a prerequisite for existing weak completeness results for coalgebraic modal logic [16, 5, 11], has played a crucial role. In particular, we have proved that every coalgebraic modal logic admits a reflexive axiomatization by axioms of rank 1; this not only means that the mentioned completeness results are, in principle, always applicable, but also implies a finite model property which states that all satisfiable formulae can be satisfied in a finite model whose size is exponentially bounded by the size of the formula.

We have then described a generic decision procedure for satisfiability in coalgebraic modal logic, assuming a decision procedure for the rather simpler *one-step satisfiability* problem. We have thus proved decidability for a wide range of modal logics, including graded and probabilistic modal logic. This goes significantly beyond the decidability result of [16], which applies only to signature functors that preserve finite sets, such as the powerset functor (whose coalgebras are standard Kripke frames). Moreover, assuming a mild complexity bound (*NP*) for one-step satisfiability, we have established a first general complexity bound for coalgebraic modal logic (*NEXPTIME*). This result applies to both graded and probabilistic modal logic; while for graded modal logic, a better bound (*PSPACE*) is known, no complexity bound at all has, to our knowledge, so far been given for probabilistic modal logic. Forthcoming work [24] will establish, under additional assumptions, tighter generic bounds which in particular push the bound for PML at least to *EXPTIME* and likely to *PSPACE*.

Acknowledgements

The author wishes to thank Till Mossakowski, Markus Roggenbach, and Horst Reichel for collaboration on CoCasl, Erwin R. Catesbeiana for favoring empty coalgebras, and Dirk Pattinson for useful discussions.

References

[1] F. Bartels, A. Sokolova, and E. de Vink. A hierarchy of probabilistic system types. In *Coalgebraic Methods in Computer Science*, volume 82 of *ENTCS*. Elsevier, 2003.
[2] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge, 2001.
[3] J. W. Carlyle and A. Paz. Realizations by stochastic finite automata. *J. Comput. System Sci.*, 5:26–40, 1971.
[4] B. Chellas. *Modal Logic*. Cambridge, 1980.
[5] C. Cîrstea and D. Pattinson. Modular construction of modal logics. In *Concurrency Theory*, volume 3170 of *LNCS*, pages 258–275. Springer, 2004.

[6] G. D'Agostino and A. Visser. Finality regained: A coalgebraic study of Scott-sets and multisets. *Arch. Math. Logic*, 41:267–298, 2002.

[7] A. Heifeitz and P. Mongin. The modal logic of probability. In *Theoretical Aspects of Rationality and Knowledge*, pages 175–186. Morgan Kaufmann, 1998.

[8] A. Heifetz and P. Mongin. Probabilistic logic for type spaces. *Games and Economic Behavior*, 35:31–53, 2001.

[9] M. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency. *J. ACM*, 32:137–161, 1985.

[10] B. Jacobs. Towards a duality result in the modal logic of coalgebras. In *Coalgebraic Methods in Computer Science*, volume 33 of *ENTCS*. Elsevier, 2000.

[11] C. Kupke, A. Kurz, and D. Pattinson. Algebraic semantics for coalgebraic logics. In *Coalgebraic Methods in Computer Science*, volume 106 of *ENTCS*, pages 219–241. Elsevier, 2004.

[12] A. Kurz. Specifying coalgebras with modal logic. *Theoret. Comput. Sci.*, 260: 119–138, 2001.

[13] K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inform. Comput.*, 94:1–28, 1991.

[14] T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-co-algebraic specification in CoCasl. *J. Logic Algebraic Programming*. To appear.

[15] D. Pattinson. Semantical principles in the modal logic of coalgebras. In *Symposium on Theoretical Aspects of Computer Science*, volume 2010 of *LNCS*, pages 514–526. Springer, 2001.

[16] D. Pattinson. Coalgebraic modal logic: Soundness, completeness and decidability of local consequence. *Theoret. Comput. Sci.*, 309:177–193, 2003.

[17] D. Pattinson. Expressive logics for coalgebras via terminal sequence induction. *Notre Dame J. Formal Logic*, 45:19–33, 2004.

[18] S. Popkorn. *First Steps in Modal Logic*. Cambridge, 1994.

[19] M. Rößiger. Coalgebras and modal logic. In *Coalgebraic Methods in Computer Science*, volume 33 of *ENTCS*. Elsevier, 2000.

[20] J. Rothe, H. Tews, and B. Jacobs. The Coalgebraic Class Specification Language CCSL. *J. Universal Comput. Sci.*, 7:175–193, 2001.

[21] J. Rutten. Universal coalgebra: A theory of systems. *Theoret. Comput. Sci.*, 249:3–80, 2000.

[22] A. Schrijver. *Theory of linear and integer programming*. Wiley Interscience, 1986.

[23] L. Schröder. Expressivity of coalgebraic modal logic: the limits and beyond. In *Foundations of Software Science And Computation Structures*, volume 3441 of *LNCS*, pages 440–454. Springer, 2005.

[24] L. Schröder and D. Pattinson. *PSPACE* reasoning for coalgebraic modal logic. In preparation.

[25] S. Tobies. *PSPACE* reasoning for graded modal logics. *J. Logic Computation*, 11:85–106, 2001.

[26] P. Turakainen. On the minimization of linear space automata. *Ann. Acad. Sci. Fenn. Ser. A I*, 506, 1972. 15 pp.

[27] I. Viglizzo. Final sequences and final coalgebras for measurable spaces. In *Algebra and Coalgebra in Computer Science*, volume 3629 of *LNCS*, pages 395–407. Springer, 2005.

Presenting Functors by Operations and Equations

Marcello M. Bonsangue[1,⋆] and Alexander Kurz[2,⋆⋆]

[1] LIACS, Leiden University, The Netherlands
[2] Department of Computer Science, University of Leicester, UK

Abstract. We take the point of view that, if transition systems are coalgebras for a functor T, then an adequate logic for these transition systems should arise from the 'Stone dual' L of T. We show that such a functor always gives rise to an 'abstract' adequate logic for T-coalgebras and investigate under which circumstances it gives rise to a 'concrete' such logic, that is, a logic with an inductively defined syntax and proof system. We obtain a result that allows us to prove adequateness of logics uniformly for a large number of different types of transition systems and give some examples of its usefulness.

1 Introduction

The question we are concerned with in this paper is how to associate to a given type of transition systems an adequate (modal) logic. Here *adequate* means that the logic is sound and complete and that two states are bisimilar iff they are logically equivalent (ie, iff they have the same theory). For the latter property, we also say that the logic is expressive or that the semantics is fully abstract.

Our starting point is the theory of coalgebras as in Rutten [29]. That is, the *type* of a category of transition systems is given by a functor T on a category $\mathcal{X}$ and transition systems of type T are T-coalgebras, ie arrows $X \to TX$ in $\mathcal{X}$. The basic idea of our approach is that an adequate logic for T-coalgebras is given by the dual functor L of T on the Stone dual $\mathcal{A}$ of $\mathcal{X}$ as explained below.

$$T \circlearrowleft \mathcal{X} \underset{S}{\overset{P}{\rightleftarrows}} \mathcal{A} \circlearrowright L \tag{1}$$

$\mathcal{A}$ is a category of algebras such as Boolean algebras or distributive lattices representing a propositional logic such as classical or positive propositional logic. P and S are the contravariant[1] functors that provide the dual equivalence between $\mathcal{X}$ and $\mathcal{A}$. Intuitively, P maps a state space X to the logic of propositions on X and S maps an algebra to its 'canonical model'. That L and T are dual means that there is an isomorphism $LP \to PT$.

The main contribution (Section 2) of the paper is the notion of a functor having a *presentation by operations and equations*. Theorem 15 shows that the category of

⋆ Supported by a fellowship of the Royal Netherlands Academy of Arts and Sciences.
⋆⋆ Partially supported by the Nuffield Foundation Grant NUF-NAL04.

[1] Given categories $\mathcal{C}, \mathcal{C}'$, we use the notation $\mathcal{C} \to \mathcal{C}'$ for (covariant) functors such as T and L as well as for contravariant functors such as P and S.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 172–186, 2006.
© Springer-Verlag Berlin Heidelberg 2006

algebras for such a functor is an equationally definable class of algebras (over Set). We then go on to show (Section 4) that, although the dual L of T gives rise to an adequate logic for T-coalgebras, the resulting logic is too abstract to be useful. In particular, inductively defined formulae, a logical calculus and a notion of (inter)derivability are still missing. But these are provided (Section 5) by a presentation of L by operations and equations. Theorem 27 shows how the operations give rise to modal operators, the equations to axioms and how the modal calculus is inherited from equational logic.

The notion of a functor being presentable by operations and equations is modelled on the notion of an algebra being presentable by generators and relations (reviewed in Section 2.1). This will be discussed in more detail now.

Comparison with Work in Domain Theory. The prototypical transition systems, called Kripke frames in modal logic, consist of just a set X and a relation R on X. Writing $\mathcal{P}$ for the operation that maps a set to its powerset, (X, R) can also be described by its 'successor map', or $\mathcal{P}$-coalgebra, $X \to \mathcal{P}X$. $\mathcal{P}$ is a functor on the category Set of sets and functions and has analogues on many categories of topological spaces, including important categories of domains. $\mathcal{P}$ is then called powerspace, hyperspace or powerdomain.

It is well-known in domain theory that the dual of the powerspace on the corresponding category of algebras can be described using modalities $\Box$ and $\Diamond$. This goes back to Johnstone [12, 13] where a dual of the powerspace, called Vietoris locale, is described. In case of distributive lattices [13, Section 1.8] anticipates the axiomatisation of positive modal logic by Dunn [10]. Winskel [32] used modalities to describe the powerdomain and Robinson [25] established the connection between the work in domain theory and that of Johnstone. Abramsky [1] extended these ideas to give logical descriptions of domains for a large number of other type constructors.

To position our contribution it will be useful to briefly summarise the work mentioned in the previous paragraph using our notation from Diagram (1). To describe a powerspace T on a domain—or more generally on a topological space—$X \in \mathcal{X}$, one has to describe the effect of T on the topology of X. This can be done without reference to X, using only the algebraic properties of the topology of X: the topology of TX is given (up to canonical iso) by the algebra $L(PX)$ which is freely generated by symbols $\Box a$, $\Diamond a$, $a \in PX$ satisfying some 'relations' as eg $\Box(a \wedge a') = \Box a \wedge \Box a'$. To summarise, a logic for T-coalgebras is obtained by describing the dual LPX of TX using the *technique of generators and relations*: The modal operators arise from the generators and the axioms from the relations.

Our paper formalises the move from generators and relations to the modal logic. Let us explain what needs to be done. Going back to the example of the powerspace, we observe that $\Box a$ is a formal symbol as a generator of LPX, but $\Box$ is a unary operator in the modal logic. Similarly, as a 'relation' $\Box(a \wedge a') = \Box a \wedge \Box a'$ is a pair of terms, but as a logical axiom it is an equation where a, a' are variables. Moreover, but related, a logic is obtained from a presentation of LPX only if the presentation does not depend on X, that is LPX is given by the 'same' generators and relations for all X. The necessary step is to generalise the notion of an algebra being presented to the notion of a functor being presented. As a consequence we obtain Theorems 15 and 27, which uniformly account for a large number of categories $\mathcal{X}$ and functors T.

Comparison with Work on Coalgebras. Work on modal logic and coalgebras started with Moss [21] whose proposal works essentially for any functor T (on the category of sets), but does not provide the linguistic means to decompose the structure of T which is needed to allow for a flexible specification language. To address this issue, subsequent work as eg [19, 28, 11] restricted attention to particular classes of functors. Pattinson [23] showed that these languages arise from modal operators given by certain natural transformations, called predicate liftings. [16] showed that, furthermore, these languages correspond to functors L on the category BA of Boolean algebras. Here we address the opposite question of how to associate a logic to a functor L.

This paper can also be seen as a sequel to [7], where we proposed a general framework for logics of coalgebras based on Stone duality. A general adequateness result was proved for what we call here abstract logics and then, studying the case of the compact powerspace $\mathcal{P}_c$, it was shown how to systematically obtain logics for coalgebras over different base categories by presenting the dual of $\mathcal{P}_c$ by generators and relations. But a formal description of the step from generators and relations to modal operators and axioms was left open.

2 Presenting Functors by Operation and Equations

This section defines what it means for a functor L on an algebraic category $\mathcal{A}$ to have a presentation by operations and equations. It is shown that, if L has such a presentation, the category $\mathsf{Alg}(L)$ of L-algebras is isomorphic to a category $\mathsf{Alg}(\Sigma, E)$ of algebras for a signature and equations and, moreover, Σ and E are obtained from the presentations of $\mathcal{A}$ and L in a modular way. Let us emphasise that this is known for the case $\mathcal{A} = \mathsf{Set}$, the novelty here coming from the need to consider other base categories than Set.

2.1 A Brief Review of Algebras and Presentations

Algebras. Given a functor L on a category $\mathcal{A}$, an L-algebra (notation: (A, α) or just α) is an arrow $\alpha : LA \to A$. A morphism $f : \alpha \to \alpha'$ is an arrow $f : A \to A'$ such that $f \circ \alpha = \alpha' \circ Lf$.

The category of algebras for a signature Σ and equations E is defined as usual[2] and denoted by $\mathsf{Alg}(\Sigma, E)$. We say that a category $\mathcal{A}$, equipped with a forgetful functor $U : \mathcal{A} \to \mathsf{Set}$, has a *presentation* (over Set) if there exists a signature Σ and equations E such that $\mathcal{A}$ is concretely[3] isomorphic to $\mathsf{Alg}(\Sigma, E)$. $\mathcal{A}$ (or more precisely $U : \mathcal{A} \to \mathsf{Set}$) is *monadic* iff $\mathcal{A}$ has such a presentation and $U : \mathcal{A} \to \mathsf{Set}$ has a left adjoint (ie free algebras exist). The left adjoint of U is denoted by F throughout (FX is the free algebra over X and UFX is the set of terms over X quotiented by the equations). Examples: The category of complete Boolean algebras has a presentation but is not monadic, whereas the category of complete atomic Boolean algebras is monadic (and dually equivalent to Set); see [12].

[2] Carriers are sets and arities may be arbitrary cardinals; we allow a set of operations for each arity and a set of equations for each set of variables.

[3] *Concretely* means that the isomorphism preserves the underlying sets.

Presenting Algebras by Generators and Relations. The following is tailored towards Section 2.2, for more see Vickers [31]. Suppose we have a monadic functor $U : \mathcal{A} \to \mathsf{Set}$ with left-adjoint F. Then the counit

$$\varepsilon_A : FUA \to A$$

gives us a canonical (albeit not economical) presentation of A, namely generated by the elements of UA and quotiented by the kernel $\{(t, s) \mid \varepsilon_A(t) = \varepsilon_A(s)\}$ of ε_A. These presentations are useful to describe operations on algebras. Unfortunately we have only space for one example.

Example 1 (modal algebras). A modal algebra, or Boolean algebra with operator (BAO), is the algebraic structure required to interpret (classical) modal logic which consists of propositional logic plus a unary modal operator $\Box$ preserving finite conjunctions. Modal algebras are therefore algebras for the functor $\mathcal{V} : \mathsf{BA} \to \mathsf{BA}$, where $\mathcal{V}A$ is defined by generators $\Box a$, $a \in A$, and relations $\Box\top = \top$, $\Box(a \wedge a') = \Box a \wedge \Box a'$.

Note that in the example above, the symbol $\Box$ appears in two roles. First we said that $\Box$ is a unary operator. But when we considered $\Box a$ as a generator, '$\Box a$' was just a formal symbol. This observation will lead to Definition 6.

Remark 2. The fact that, in the example above, $\mathcal{V}$ is on BA (and not on Set) takes care of the propositional part of modal logic. The definition of $\mathcal{V}A$ can be phrased more abstractly by saying that the insertion of generators $UA \to U\mathcal{V}A, a \mapsto \Box a$ is a universal finite-meet preserving function, that is,

$$\begin{array}{ccc} UA & \longrightarrow & U\mathcal{V}A \\ & {\scriptstyle f}\searrow & \vdots\, {\scriptstyle Uf^\sharp} \\ & & UB \end{array}$$

for all $B \in \mathsf{BA}$ and all finite-meet preserving functions $f : UA \to UB$ there is a unique Boolean algebra morphism $f^\sharp : \mathcal{V}A \to B$ with $Uf^\sharp(\Box a) = f(a)$. From this observation it is straightforward to show that $\mathsf{Alg}(\mathcal{V})$ is indeed isomorphic to the category of modal algebras as usually defined (see eg [15, Definition 2.2.2], [4, Definition 5.19]).

Definition 3 (presentation by generators and relations). Let $U : \mathcal{A} \to \mathsf{Set}$ be monadic (see p.174) with left adjoint F. A *presentation* $\langle G, R\rangle$ consists of a set of 'generators' G and a set of 'relations' $R \subseteq UFG \times UFG$.

Definition 4 (presented algebra). Continuing from the previous definition, a morphism $f : FG \to B$ in $\mathcal{A}$ *satisfies the relations* R if $(t, s) \in R \Rightarrow Uf(t) = Uf(s)$. An algebra A is *presented* by $\langle G, R\rangle$ if

$$\begin{array}{ccc} FG & \xrightarrow{q} & A \\ & {\scriptstyle f}\searrow & \vdots\, {\scriptstyle f^+} \\ & & B \end{array}$$

- A comes with an insertion of generators $q : G \to UA$ (or, equivalently, $q : FG \to A$) satisfying the relations R,
- for all $B \in \mathcal{A}$ and all $f : FG \to B$ satisfying the relations R there is a unique $f^+ : A \to B$ with $f^+ \circ q = f$.

Proposition 5. Every presentation presents an algebra.

Proof. The proof relies on the fact that, as a category monadic over Set, $\mathcal{A}$ has co-equalisers. The object presented by $\langle G, R\rangle$ is given by the coequaliser

$$FR \overset{\pi_1^\sharp}{\underset{\pi_2^\sharp}{\rightrightarrows}} FG \xrightarrow{q} A.$$

where $\pi_1^\sharp, \pi_2^\sharp$ come from the projections $\pi_1, \pi_2 : R \to UFG$. More concretely, q is the quotient wrt the smallest congruence containing R.

2.2 Presenting Functors by Operations and Equations

Example 1 above shows how the functor $\mathcal{V} : \mathsf{BA} \to \mathsf{BA}$ is described using generators and relations. In order to obtain a modal logic from that description, one has to upgrade the set of formal symbols $\Box a$ to a unary operator $\Box$ and, similarly, the relations $\Box(a \wedge a') = \Box a \wedge \Box a'$ to equations in variables a, a' (Definition 6). Moreover, the presentation of $\mathcal{V}A$ is the 'same' for all A. This will be crucial for the move from a presentation to a logic: The modal operators (ie the generators) and the axioms (ie the relations) should depend only on the functor and not on specific algebras (Definition 7).

Definition 6 (presentation by operations and equations). Let $U : \mathcal{A} \to \mathsf{Set}$ be monadic with left adjoint F. A *presentation of a functor* $L : \mathcal{A} \to \mathcal{A}$ by operations and equations consists of

1. a set Σ of operations $\sigma \in \Sigma$ with arities n_σ which gives rise to a functor $G_\Sigma : \mathsf{Set} \to \mathsf{Set}$, $X \mapsto \coprod_{\sigma \in \Sigma} X^{n_\sigma}$,
2. a class $\mathcal{C}$ of sets (of variables) and a collection $E = (E_V)_{V \in \mathcal{C}}$ of equations $E_V \subseteq (UFG_\Sigma UFV)^2$.

A presentation is called *finite* if Σ only contains operations of finite arity and $\mathcal{C}$ contains only finite sets.

Definition 7 (presented functor). Continuing from the previous definition, a morphism $f : FG_\Sigma UA \to B$ *satisfies the equations* E if for all $V \in \mathcal{C}$ and all $v : FV \to A$ it holds $(t, s) \in E_V \;\Rightarrow\; (f \circ FG_\Sigma Uv)(t) = (f \circ FG_\Sigma Uv)(s)$. A natural transformation $f : FG_\Sigma U \to L$ satisfies the equations if f_A satisfies the equations for all $A \in \mathcal{A}$.

A functor L is *presented* by $\langle \Sigma, E\rangle$ if

$$FG_\Sigma UFV \xrightarrow{FG_\Sigma Uv} FG_\Sigma UA \xrightarrow{q_A} LA, \qquad FG_\Sigma UA \xrightarrow{f} B, \qquad LA \overset{f^+}{\dashrightarrow} B$$

– L comes with a natural transformation, called insertion of generators, $q : G_\Sigma U \to UL$ (or, equivalently, $q : FG_\Sigma U \to L$) satisfying the equations E,
– for any $B \in \mathcal{A}$ and morphism $f : FG_\Sigma UA \to B$ satisfying the equations E there is a unique morphism $f^+ : LA \to B$ such that $f^+ \circ q_A = f$.

Remark 8. 1. Roughly speaking, if $\mathcal{A}$ represents a finitary logic, we will need $\mathcal{C}$ to only contain finite sets of variables. But for infinitary logics, a typical requirement for an operator $\Box$ in Σ would be to preserve all meets, which is expressed by equations $\bigwedge_{v \in V} \Box v = \Box \bigwedge_{v \in V} v$ where V runs through all cardinals.
2. Here we explain the format of the equations $E_V \subseteq (UFG_\Sigma UFV)^2$.
   – $E_V \subseteq (UFG_\Sigma UFV)^2$ means that the terms appearing in equations may freely use the operations for $\mathcal{A}$ but do not contain nested occurrences of operations from Σ.
   – Intuitively, this format arises from our interests in logics that describe coalgebras for a *functor* T. In contrast to coalgebras for a comonad, the coalgebra map $\xi : X \to TX$ encodes what the transition system (X, ξ) can perform in *one* step. From this point of view, the format $E_V \subseteq (UFG_\Sigma UFV)^2$ of the axioms E is not a restriction, but formalises that we do *not need* nested modalities to describe a single transition (nested modalities describe sequences of transition steps).
   – Technically, equations of the form $E_V \subseteq (UFG_\Sigma UFV)^2$ suffice since the terms to be quotiented are all in $UFG_\Sigma UA$. The reason to exclude more general equations is Theorem 15.
3. Using the approach of Linton [20] and Rosický [26], one can show that $\mathsf{Alg}(L)$ can always be described by operations and equations over $\mathcal{A}$. But in that approach the arities describing $\mathsf{Alg}(L)$ are objects in $\mathcal{A}$ and not cardinalities in Set. In other words, what our approach adds here is that under Definition 7 one does obtain a presentation not only of $\mathsf{Alg}(L)$ over $\mathcal{A}$ but also of $\mathsf{Alg}(L)$ over Set (Theorem 15). The latter is essential to get logics as in Section 5.
4. A presentation of $\mathsf{Alg}(L)$ over Set cannot be expected to exist in general, because monadic functors are not closed under composition: Even if $\mathsf{Alg}(L) \to \mathcal{A}$ and $\mathcal{A} \to \mathsf{Set}$ are monadic, the composition $\mathsf{Alg}(L) \to \mathsf{Set}$ need not be so. This is discussed in detail in Kelly and Power [14]. For our purposes, our approach has the advantage that we obtain a presentation of $\mathsf{Alg}(L)$ modularly from presentations of $\mathcal{A}$ and L (Theorem 15). Moreover, we do not insist on $\mathsf{Alg}(L)$ having free algebras.
5. Example 1 seems to suggest that one could express the interplay of the modal and Boolean operators by a distributive law between the functor L describing the modal operators and the monad UF describing the Boolean operators. This approach does not work as L is not even defined on underlying sets but only on algebras.
6. An approach based on monads would not be appropriate because we do not want to insist that $\mathsf{Alg}(L)$ has free algebras. (For an example where $\mathsf{Alg}(L)$ doesn't have free algebras although $\mathcal{A}$ has, take $\mathcal{A}$ to be the category of complete atomic Boolean algebras and L the dual of the powerset.)

Definition 7 will allows us to present $\mathsf{Alg}(L)$ by composing a presentation of $\mathcal{A}$ with a presentation of L, see Theorem 15. Logically, this corresponds to extending a basic

propositional logic (which presents $\mathcal{A}$) with modal operators and modal axioms (which present L). This is also the idea underlying the following examples.

Example 9. 1. As mentioned already in the introduction, to define a functor L by describing LA by generators and relations is a common technique. In all of the cited [12, 13, 25, 31] the given presentations are in fact presentations of L by operations and equations. The reason to make this notion explicit here is to have a uniform translation from presentations to logics that works for functors L in general (see Theorems 15 and 27).

2. The functor $\mathcal{V}$ of Example 1 is presented by a signature containing one unary operation $\Box$, that is, $G_\Sigma X = X$. Further, $V = \{v_0, v_1\}$, $\mathcal{C} = \{V\}$, and, writing '$\cdot = \cdot$' instead of '$(\cdot, \cdot)$', $E_V = \{\Box\top = \top, \Box(v_0 \wedge v_1) = \Box v_0 \wedge \Box v_1\}$.
3. The functor $\mathcal{V}$ above is the dual of the powerspace on Stone. Other type constructors on Stone are studied in [18] (called Vietoris polynomial functors) and their duals on BA are all presentable by operations and equations.
4. The Kripke polynomial functors on Set (including powerset) of [11] have duals on complete atomic Boolean algebras, which have a presentation. The description of the dual of the finite (or compact) powerspace on posets and sets in [7] also provides examples of presentations of functors.

Proposition 10. Each presentation presents a functor.

Proof. Given a presentation $\langle \Sigma, E \rangle$ we define the functor L on objects A as

$$FE_V \underset{\pi_2^\sharp}{\overset{\pi_1^\sharp}{\rightrightarrows}} FG_\Sigma UFV \xrightarrow{FG_\Sigma Uv} FG_\Sigma UA \xrightarrow{q_A} LA \tag{2}$$

where q_A is the joint coequaliser of all pairs $(FG_\Sigma Uv \circ \pi_1^\sharp, FG_\Sigma Uv \circ \pi_2^\sharp)$ where v ranges over arrows $FV \to A$. The universal property of LA gives the action of L on morphisms and the naturality of q_A.

Proposition 11. The functors that have a presentation are closed under composition.

Proof (Sketch). Consider L_1 and L_2 with presentations (Σ_1, E_1), (Σ_2, E_2). Then $L_1 L_2$ is presented by $(\sum_{i \in n} n_i)$-ary operations $\sigma((t_i)_{i \in n})$ where $\sigma \in \Sigma_1$ is n-ary and the t_i are n_i-ary Σ_2-terms in $UFG_{\Sigma_2}V$. The equations $s((t_i)_{i \in n}) = s'((t_i')_{i \in n})$ are all those that can be obtained from the equations $s = s'$ derivable from E_1 and then substituting terms $t_i = t_i'$ (with identities derivable from E_2).

Remark 12. The proposition shows that we can build up presentations modularly. The construction in the proof has the disadvantage though, that (many) new operations and equations have to be introduced. In practice, therefore, one would rather introduce an additional sort with the benefit of using exactly the operations and equations of the two original presentations. This is as in, eg, [1, 27, 11, 9, 30] and will be detailed elsewhere.

Although we are not interested in the case $\mathcal{A} =$ Set as such, the following shows that Definition 6 is natural: Up to a size restriction, any functor on Set has a presentation. In particular.

Proposition 13. A functor on Set has a finite presentation if and only if it is finitary.

Proof. If an endofunctor L on Set is given as in Diagram (2), then it is not difficult to show that it preserves filtered colimits, given that F, U, and G do so and the sets V are finite. Conversely, given a finitary L, we obtain Σ and E as follows [24, 1.5]. For Σ we let each element in Ln be an n-ary operation,[4] that is, $G_\Sigma X = \coprod_{n<\omega} Ln \times X^n$. Further, $V = \{v_i \mid i < \omega\}$, $\mathcal{C} = \{V\}$ and $E_V = \{(Lf(\sigma))(v_0, \dots v_{m-1}) = \sigma(v_{f(0)}, \dots v_{f(n-1)}) \mid n < \omega, \sigma \in Ln, f : n \to m\}$. The natural transformation $q_A : G_\Sigma UA \to LA$ then maps $n < \omega, \sigma \in Ln, f : n \to A$ to $Lf(\sigma) \in LA$.

If we do not insist on $\mathcal{A} =$ Set, the two notions become different as the second part of the proof does not generalise. A general characterisation of the functors having a finite presentation will be given elsewhere.

We still have to show how functors that are presentable by operations and equations give rise to logics. Since $\mathcal{A}$ is monadic over Set, we can assume that we a have a presentation of $\mathcal{A}$ as a category of algebras $\mathsf{Alg}(\Sigma_\mathcal{A}, E_\mathcal{A})$ given by a signature $\Sigma_\mathcal{A}$ and equations $E_\mathcal{A}$.

Definition 14 **($\Sigma_\mathcal{A} + \Sigma_L, E_\mathcal{A} + E_L$).** Let $\mathcal{A} \cong \mathsf{Alg}(\Sigma_\mathcal{A}, E_\mathcal{A})$ and $L : \mathcal{A} \to \mathcal{A}$ be presented by $\langle \Sigma_L, E_L \rangle$. Denote by $\Sigma_\mathcal{A} + \Sigma_L$ the disjoint union of the signatures and by $E_\mathcal{A} + E_L$ the disjoint union where equations in $E_\mathcal{A}$ and E_L are understood as equations over $\Sigma_\mathcal{A} + \Sigma_L$.[5]

$E_\mathcal{A} + E_L$ is a sound and complete (equational) logic for L-algebras:

Theorem 15. *Let $\mathcal{A} \cong \mathsf{Alg}(\Sigma_\mathcal{A}, E_\mathcal{A})$ be monadic and $\langle \Sigma_L, E_L \rangle$ a presentation of $L : \mathcal{A} \to \mathcal{A}$. Then $\mathsf{Alg}(\Sigma_\mathcal{A} + \Sigma_L, E_\mathcal{A} + E_L)$ is isomorphic to $\mathsf{Alg}(L)$.*

Proof (Sketch). Write $\Sigma = \Sigma_\mathcal{A} + \Sigma_L$, $E = E_\mathcal{A} + E_L$. Consider $\alpha : LA \to A$. The corresponding Σ-algebra A has carrier UA and the interpretation σ^A of operations $\sigma \in \Sigma_L$ is given by $(UA)^{n_\sigma} \to UFG_{\Sigma_L}UA \overset{Uq_A}{\to} ULA \overset{U\alpha}{\to} UA$. A satisfies the equations $E_\mathcal{A}$ because A does. A satisfies the equations E_L because q_A does (Definition 7) and because of the format $(E_L)_V \subseteq (UFG_{\Sigma_L}UFV)^2$.

Conversely, every (Σ, E)-algebra A is also an algebra in $\mathcal{A}$. We then obtain, from the operations in Σ_L, a function $G_{\Sigma_L}UA \to UA$, ie a morphism $f : FG_{\Sigma_L}UA \to A$. Since A satisfies the equations E_L we obtain by the universal property of Definition 7 a morphism $\alpha = f^+ : LA \to A$.

Remark 16.
1. Without requiring $(E_L)_V \subseteq (UFG_{\Sigma_L}UFV)^2$ in Definition 7, L-algebras would not need to satisfy the equations E_L.
2. We do not insist that $\mathsf{Alg}(L) \to$ Set be monadic (Remark 8.6).

[4] Writing Ln we assume that n is the set $\{0, \dots n-1\}$.

[5] Strictly speaking, E_L was defined on equivalence classes of $\Sigma_\mathcal{A}$-terms. Formally, one obtains the new E_L, denoted E'_L, as follows. Let $T_{\Sigma_\mathcal{A}}V$ be the set of $\Sigma_\mathcal{A}$-terms over V. Consider a half-inverse m of the quotient $T_{\Sigma_\mathcal{A}}G_{\Sigma_L}T_{\Sigma_\mathcal{A}}V \to UFG_{\Sigma_L}UFV$ (m chooses a representative for each equivalence class). Then $E'_L = \{(m(t), m(s)) \mid (t, s) \in E_L\}$.

3. Assume that $\mathcal{A} = \mathsf{Set}$ and that L has a finitary presentation. Then Theorem 15 and Proposition 13 specialise to the well-known result that $\mathsf{Alg}(L)$ is a variety that can be described by equations without nesting of operation symbols, see [3, Section III.3.2, III.4.3]. Let us remark that [3] does not have the notion of a presentation of a functor. We need it here to generalise from Set to other monadic categories $\mathcal{A}$.

3 A Brief Review of Coalgebras and Stone Duality

Coalgebras. Given a functor T on a category $\mathcal{X}$, a T*-coalgebra* (notation: (X, ξ) or just ξ) is an arrow $\xi : X \to TX$ in $\mathcal{X}$. A morphism $f : \xi \to \xi'$ is an arrow $f : X \to X'$ such that $Tf \circ \xi = \xi' \circ f$.

Throughout the paper it will be the case that $\mathcal{X}$ is the category Set of sets and functions or some category of topological spaces or domains. It makes therefore sense to speak of the elements, or *states*, of some $X \in \mathcal{X}$. We say that two states x, x' of $\xi : X \to TX$ and $\xi' : X' \to TX'$ are *behaviourally equivalent* or *bisimilar* if there are coalgebra morphisms f, f' with $f(x) = f(x')$. This notion of bisimilarity agrees with the standard one in all cases we are aware of.

Stone Duality. We sketch some background on Stone duality. It may be skipped and consulted later. A *topological space* $(X, \mathcal{O})$ is a set X together with a collection $\mathcal{O}$ of subsets of X closed under finite intersections and arbitrary unions. Elements $a \in \mathcal{O}$ are called *open* sets. A function $(X, \mathcal{O}) \to (X', \mathcal{O}')$ is *continuous* if f^{-1} preserves opens, that is, restricts to a map $\mathcal{O}' \to \mathcal{O}$. Topological spaces and continuous maps form the category Top. Note that f^{-1} preserves finite intersections and arbitrary unions.

Abstracting from the set of points X and axiomatising the algebraic properties of a topology $\mathcal{O}$, one arrives at the following notion. A *frame*[6] A is a distributive lattice (with bottom $\bot$ and top $\top$) with infinite joins satisfying the infinite distributive law $a \wedge \bigvee C = \bigvee\{a \wedge c \mid c \in C\}$ for all $a \in A$ and all subsets $C \subseteq A$. Frames with functions preserving arbitrary joins and finite meets form the category Frm. Frm has free algebras, in other words, the forgetful functor from Frm to Set mapping each frame to its underlying set is monadic.

There are contravariant functors

$$\mathsf{Top} \underset{S}{\overset{P}{\rightleftarrows}} \mathsf{Frm}$$

$P(X, \mathcal{O}) = \mathcal{O}$, $P(f) = f^{-1}$ ('P' since P associates to a space X the algebra of *p*redicates over X). If $(X, \mathcal{O})$ is a discrete topological space, then P is the contravariant powerset functor. $S(A) = \mathsf{Frm}(A, 2)$, where 2 is the two element frame (consisting of $\bot, \top$). $S(A)$ carries the topology generated by the sets, for each $a \in A$, $\{s \in S(A) \mid s(a) = \top\}$. $S(f) = \lambda s \in S(A) \,.\, s \circ f$. For example, if A is a Boolean algebra, then SA is the space of ultrafilters over A (ultrafilters represent maximal consistent theories).

[6] The notions 'frame' and 'Kripke frame' come from different areas are not related.

Fact 17. *P, S are adjoint on the right, that is, there is a bijection, natural in X and A,*

$$\mathsf{Top}(X, SA) \cong \mathsf{Frm}(A, PX).$$

The adjunction restricts to a dual equivalence on the subcategories of spaces X and frames A for which the units $X \to SPX$ and $A \to PSA$ are isomorphisms. These spaces and frames are called *sober* and *spatial*, respectively. We will need later that a frame A is spatial iff

$$\forall a, a' \in A.(a \not\leq a' \;\Rightarrow\; \exists s \in SA.(s(a) = \top \;\&\; s(a') = \bot)). \tag{3}$$

The dual equivalence of sober spaces and spatial frames can be restricted to obtain a large number of interesting examples. We mention here only the duality of the categories Stone of Stone spaces and BA of Boolean algebras and the duality of the categories Spec of spectral spaces and DL of distributive lattices. For details and more examples see [12, 31, 2].

The adjunction can also be 'upgraded' to an adjunction between Top and OFrm, the category of observation frames [5]. It restricts to a dual equivalence for all T0-spaces. We can then include the category of posets into the list of possible topological spaces and treat propositional logics without negation but with infinitary meets [6]. This approach was also used in [7].

4 Abstract Logics for Coalgebras

It is shown that adequate logics for T-coalgebras are given by the functor L that is dual to T. This section is independent of Section 2.

Definition 18 (dual functor). Let $P : \mathcal{X} \to \mathcal{A}$ and $S : \mathcal{A} \to \mathcal{X}$ be a dual equivalence and T a functor on $\mathcal{X}$. (L, δ), or simply L, is called a (or the) dual of T on $\mathcal{A}$ if there is a natural isomorphism $\delta : LP \to PT$.

All duals of T are naturally isomorphic and the canonical one is PTS (but more interesting are those duals L that have a purely algebraic description (Definition 7) which does not go via $\mathcal{X}$). δ allows us to consider the collection of predicates on a coalgebra as an L-algebra. That is, we can lift the functors P and S to an equivalence of algebras and coalgebras. Explicitly, on objects, the lifted $\tilde{P}$ and $\tilde{S}$ are given as

$$\begin{aligned} \tilde{P}(X, \xi) \;&=\; LPX \xrightarrow{\delta_X} PTX \xrightarrow{P\xi} PX \\ \tilde{S}(A, \alpha) \;&=\; SA \xrightarrow{S\alpha} SLA \cong SLPSA \xrightarrow{(S\delta S)_A} SPTSA \cong TSA \end{aligned}$$

In order to interpret the dual equivalence connecting $\mathcal{A}$ and $\mathcal{X}$ as a duality between a logical calculus and its semantics, we need to more specific. *For the remainder of the paper* we will be working in the situation described by the following diagram

$$
\begin{array}{ccccc}
 & \mathsf{Coalg}(T) & \underset{\tilde{S}}{\overset{\tilde{P}}{\rightleftarrows}} & \mathsf{Alg}(L) & \\
 & \downarrow & & \downarrow & \\
T \circlearrowright & \mathcal{X} & \underset{S}{\overset{P}{\rightleftarrows}} & \mathcal{A} & \circlearrowleft L \\
 & \downarrow & & F \uparrow \downarrow U & \\
 & \mathsf{Set} & & \mathsf{Set} &
\end{array}
\tag{4}
$$

where we assume that

- the dual equivalence between $\mathcal{X}$ and $\mathcal{A}$ arises from the adjunction of Top and Frm (or Top and OFrm) by restricting to subcategories (see Section 3),
- L is dual to T (see Definition 18),
- $\mathcal{A}$ is monadic (see p.174),
- $\mathsf{Alg}(L) \to \mathcal{A}$ has a left adjoint (ie $\mathsf{Alg}(L)$ has free algebras).

Let us emphasise that the last requirement is not essential [7]. But it simplifies the presentation considerably, as we can now take the initial algebra in $\mathsf{Alg}(L)$ as a canonical set of propositions. We consider this algebra of propositions as an *abstract logic for T-coalgebras*, see Definition 21.

Definition 19 ($\mathsf{Prop}(Var)$). Denote by $\mathsf{Prop}(Var)$ the (carrier of the) free L-algebra over Var and call the elements of $\mathsf{Prop}(Var)$ propositions over variables in Var.

The algebraic semantics is defined in the usual way. Recall that there is a bijection between functions $Var \to UA$ and morphisms $\mathsf{Prop}(Var) \to A$.

Definition 20 (algebraic semantics). The algebraic semantics $\varphi^{A,h}$ of $\varphi \in \mathsf{Prop}(Var)$ wrt an algebra $A \in \mathsf{Alg}(L)$ and a valuation of variables $h : Var \to UA$ is $\varphi^{A,h} = h^{\#}(\varphi)$ where $h^{\#} : \mathsf{Prop}(Var) \to A$ is the unique extension of h. $\mathsf{Alg}(L) \models (\varphi \leq \psi)$ if $\varphi^{A,h} \leq \psi^{A,h}$ for all algebras A and all $h : Var \to UA$.

To each coalgebra (X, ξ) we can associate via $\tilde{P}$ the algebra of propositions over X. This gives the coalgebraic semantics.

Definition 21 (coalgebraic semantics). The semantics $[\![\varphi]\!]_{(X,\xi,h)}$ of a formula $\varphi \in \mathsf{Prop}(Var)$ wrt a coalgebra $(X, \xi) \in \mathsf{Coalg}(T)$ and a valuation $h : Var \to PX$ is given by $[\![\varphi]\!]_{(X,\xi,h)} = \varphi^{\tilde{P}(X,\xi),h}$. We write $\mathsf{Coalg}(T) \models (\varphi \vdash \psi)$ if $[\![\varphi]\!]_{(X,\xi,h)} \subseteq [\![\psi]\!]_{(X,\xi,h)}$ for all coalgebras and all valuations.

The following proposition can be extended to account for propositions with variables if the notion of bisimulation is appropriately adapted. But we will restrict ourselves to

Proposition 22. Propositions in $\mathsf{Prop}_\emptyset$ are invariant under bisimilarity.

Proof. We have to show that, given a coalgebra morphism $f : (X, \xi) \to (X', \xi')$ and $x \in X$, that $x \in [\![\varphi]\!]_{(X,\xi)} \Leftrightarrow f(x) \in [\![\varphi]\!]_{(X',\xi')}$. This follows directly from the universal property of the initial algebra $\mathsf{Prop}_\emptyset$.

The essence of completeness wrt to the coalgebraic semantics is:

Proposition 23. $\mathsf{Alg}(L) \models (\varphi \leq \psi) \;\; \Leftrightarrow \;\; \mathsf{Coalg}(T) \models (\varphi \vdash \psi)$.

Proof. ‘ $\Rightarrow$ ’ (soundness) is immediate. For ‘ $\Leftarrow$ ’ (completeness) assume $\not\models_{\mathsf{Alg}(L)} \varphi \leq \psi$, ie $\varphi \not\leq \psi$ in $\mathsf{Prop}(\mathit{Var})$. Since $\mathsf{Prop}(\mathit{Var})$ is spatial (see (3)), there is x in $\tilde{S}(\mathsf{Prop}(\mathit{Var}))$ such that $x \in [\![\varphi]\!]_{\mathsf{Prop}(\mathit{Var})}$ and $x \notin [\![\psi]\!]_{\mathsf{Prop}(\mathit{Var})}$.

Proposition 24. $\mathsf{Prop}_\emptyset$ is expressive. That is, if two elements x, x' of two coalgebras (X, ξ), (X', ξ') are not bisimilar, then there is $\varphi \in \mathsf{Prop}_\emptyset$ such that $x \models \varphi \;\; \Leftrightarrow \;\; x' \not\models \varphi$.

Proof. Without loss of generality, let us assume that x, x' are two different elements of the final coalgebra (Z, ζ). The two points can be distinguished by a proposition since $\mathsf{Prop}_\emptyset \to \tilde{P}(Z, \zeta)$ is surjective and PZ is a T0-space.

To summarise the section, we have seen how to obtain an adequate logic for T-coalgebras (where T is an arbitrary functor on a category $\mathcal{X}$ satisfying the conditions summarised under Diagram (4)): Just consider as formulae the elements of the initial L-algebra where L is the dual of T. We called this logic abstract as these formulae do not have much structure. For example, modal operators, an explicit inductive construction of the set of formulae and a logical calculus are still missing.

5 Concrete Logics for Coalgebras

We can now combine the abstract logics from Section 4 with the presentations of functors of Section 2. Assuming that L has a presentation, Theorem 15 gives us an equational calculus for $\mathsf{Alg}(L)$. Via the coalgebraic semantics of Definition 21 this yields an equational logic for T-coalgebras, which is adequate by Propositions 22 to 24 and concrete in the sense that we have the equational calculus for reasoning about the coalgebras.

In this section, we translate the equational logic of Theorem 15 to a modal logic. In the case of $\mathcal{A} = \mathsf{BA}$ (which corresponds to adding a modal logic to classical propositional logic) this is particularly simple: An equation $t = s$ corresponds to the modal formula $t \leftrightarrow s$. As we are interested also in $\mathcal{A} = \mathsf{DL}$ (and various subcategories), we do not assume here that the logics have implication. We therefore use in the modal logics the notation $\varphi \vdash \psi$ to represent the algebraic $\varphi \leq \psi$. As it is clear from Definition 21, $\vdash$ corresponds to local consequence in the terminology of modal logic.

Definition 25 (modal logic for T-coalgebras). Let $\mathcal{X}$ and $\mathcal{A}$ be categories as described in Diagram (4) and T a functor on $\mathcal{X}$. Assume that the dual L of T has a presentation $\langle \Sigma_L, E_L \rangle$ (Definition 7) and let $\mathcal{A} \cong \mathsf{Alg}(\Sigma_\mathcal{A}, E_\mathcal{A})$. Operations in $\Sigma_\mathcal{A}$ are called propositional connectives and operations in Σ_L are called modal operators. Following established notation, we write modal operators $\sigma \in \Sigma_L$ as $[\sigma]$. We define

Formulae. The set of formulae over a set Var of propositional variables is the smallest set containing Var and closed under operations in $\Sigma_\mathcal{A}$ and Σ_L.
Sequents. A sequent $\varphi \vdash \psi$ consists of two formulae φ, ψ.

Axiom Schemes. Each equation $\varphi = \psi$ in $E_{\mathcal{A}}$ or E_L gives rise to axiom schemes $\varphi \vdash \psi$ and $\psi \vdash \varphi$. An axiom is obtained from an axiom scheme by replacing the variables[7] with formulae.

Calculus. We use $\varphi \dashv\vdash \psi$ as an abbreviation for $\varphi \vdash \psi$ and $\psi \vdash \varphi$. The rules have to guarantee that $\dashv\vdash$ is an equivalence relation. Moreover, for each n-ary operator $\sigma \in \Sigma_{\mathcal{A}} + \Sigma_L$, we have the congruence rule

$$\frac{\varphi_i \dashv\vdash \psi_i \quad 0 \leq i < n}{\sigma(\varphi_i) \dashv\vdash \sigma(\psi_i)}$$

Semantics. Given a coalgebra (X, ξ) and a valuation $h : \mathit{Var} \to PX$, the semantics $[\![\varphi]\!]_{(X,\xi,h)}$ of a formula is defined inductively on the structure of formulae. For an n-ary modal operator $\sigma \in \Sigma_L$ its semantics is given by (for δ see Definition 18)

$$(UPX)^n \longrightarrow G_{\Sigma_L} UPX \xrightarrow{q_{PX}} ULPX \xrightarrow{U\delta_X} UPTX \xrightarrow{UP\xi} UPX$$

mapping $([\![\varphi_i]\!]_{(X,\xi,h)})_{0 \leq i < n}$ to $[\![[\sigma](\varphi_i)]\!]$.

Remark 26 (other approaches to modal logics for coalgebras). Apart from Moss [21], all subsequent work we are aware of (as eg [28, 11, 18, 22]) can be casted in terms of so-called predicate liftings as in Pattinson [23]. Predicate liftings give semantics to modal operators for T-coalgebras. They appear here as $(UPX)^n \to G_{\Sigma_L} UPX \stackrel{q_{PX}}{\to} ULPX \stackrel{U\delta_X}{\to} UPTX$. It was shown in [16] that any logic given by predicate liftings can be described by a functor L on BA that has a presentation. Our approach therefore subsumes existing ones. But we have also vastly generalised the previous work by moving from set to other topological categories and from BA to other algebraic categories. Moreover, we established a criterion for functors L to give rise to an adequate logic.

Theorem 27. *Let $\mathcal{L}$ be a logic for T-coalgebras as described in Definition 25. The formulae of $\mathcal{L}$ are invariant under bisimilarity and $\mathcal{L}$ is sound, complete and expressive.*

Proof (Sketch). First show that equational deduction is equivalent to deduction in $\mathcal{L}$. It then follows from Theorem 15 that $\mathsf{Prop}(\mathit{Var})$ is a quotient of the set of $\mathcal{L}$-formulae wrt to the interderivability $\dashv\vdash$ (the so-called Lindenbaum-Tarski algebra of $\mathcal{L}$). Moreover, the coalgebraic semantics (Definition 21) of $\mathcal{L}$ is equivalent to the one from Definition 25. Now, having established the relationship between equational and modal logic, soundness and completeness is Proposition 23. Invariance under behavioural equivalence and expressiveness are Propositions 22 and 24, respectively.

Remark 28. To keep the presentation in Section 4 simple, we assumed there that $\mathsf{Alg}(L)$ has free algebras. But, as shown in [7], this assumption is not necessary (neither always desirable: if T is the powerset functor, then $\mathsf{Coalg}(T)$ does not have a final coalgebra and $\mathsf{Alg}(L)$ does not have an initial algebra). Theorem 27 then still holds (assuming, as in [7], that T weakly preserves limits of chains).

[7] These are the variables from V, see Definition 7, which are different from the propositional variables from Var.

Example 29. 1. In the case of $\mathcal{A} = \mathsf{BA}$ one can use $\rightarrow$ instead of $\vdash$. For example, the equational logic for the functor $\mathcal{V}$ (Example 9.2) translates to a modal logic that adds to classical propositional logic the two axiom schemes $\Box\top \leftrightarrow \top$ and $\Box(v_0 \wedge v_1) = \Box v_0 \wedge \Box v_1$. This is easily seen to be equivalent to the standard calculus of modal logic.

2. The logics of [18] can be understood as presentations of the respective functors.

3. The presentations of the duals of the Kripke polynomial functors of [11] give rise to the infinitary versions of the logics studied there.

Example 30. In [7], we derived in a uniform way the logic for finitely branching transition systems on different topological spaces. The idea was to describe the dual L of the finite (= compact) powerspace, similarly to Example 1, by generators and relations. The completeness proof of the corresponding logics proceeded by, what we call here, the abstract logic (Section 4) of L. But the step from the presentation by generators and relations to the logic was not worked out, being routine and tedious. This gap can now be filled by simply appealing to Theorem 27.

6 Conclusion and Further Work

This paper introduced the notion of a functor having a presentation by operations and equations. It explains how generators and relations give rise to modal operators and axioms and leads to Theorems 15 and 27 which give automatic adequateness proofs once a presentation is given. From a mathematical point of view, the work contributes to the question when a category $\mathsf{Alg}(L)$ has a presentation by operations and equations.

Further Work. 1. The completeness result relates dual categories as eg BA and Stone or DL and Spec. How completeness wrt Set-coalgebras can be derived from these results is investigated in [17].

2. Remark 12 indicated how to compose presentations of functors. A detailed exposition of this important topic is future work.

3. Proposition 13 showed that the functors on Set with a finite presentation are precisely the finitary functors. A generalisation of this result to other monadic categories than Set will be given elsewhere.

4. An important extension will introduce presentations by operations and implications. These would be necessary to account for some of the functors in, for example, [1].

5. In [8] we apply the notion of a functor having a presentation to the extension of distributive lattices with operators. We show that presentations over posets (which amounts to moving from algebras to ordered algebras) are useful to handle monotone operators.

6. Another important extension will be to replace Set with a presheaf category $\mathsf{Set}^{\mathcal{C}}$ (which amounts essentially to moving from one-sorted to many-sorted algebras). This will allow us to treat logics with quantifiers or logics for name-passing calculi.

Acknowledgements. We would like to thank the referees for valuable suggestions. The second author profited from discussions with Neil Ghani, Clemens Kupke and Jiří Rosický.

References

1. S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51, 1991.
2. S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science*. OUP, 1994.
3. J. Adámek and V. Trnková. *Automata and Algebras in Categories*. Kluwer, 1990.
4. P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. CUP, 2001.
5. M. Bonsangue, B. Jacobs, and J. N. Kok. Duality beyond sober spaces: Topological spaces and observation frames. *Theoret. Comput. Sci.*, 151, 1995.
6. M. Bonsangue and J. Kok. Towards and infinitary logic of domains: Abramsky logic for transition systems. *Inform. and Comput.*, 155, 1995.
7. M. Bonsangue and A. Kurz. Duality for logics of transition systems. In V. Sassone, editor, *FoSSaCS'05*, volume 3441 of *LNCS*, 2005.
8. M. Bonsangue, A. Kurz, and I. Rewitzky. Coalgebraic representations of distributive lattices with operators. To appear in Topology and its Applications.
9. C. Cîrstea and D. Pattinson. Modular construction of modal logics. In *CONCUR'04*, LNCS 3170, 2004.
10. J. M. Dunn. Positive modal logic. *Studia Logica*, 55(2), 1995.
11. B. Jacobs. Many-sorted coalgebraic modal logic: a model-theoretic study. *Theor. Inform. Appl.*, 35, 2001.
12. P. Johnstone. *Stone Spaces*. Cambridge University Press, 1982.
13. P. Johnstone. Vietoris locales and localic semilattices. In *Continuous Lattices and their Applications*, volume 101 of *Lecture Notes in Pure and Applied Mathematics*. Marcel Dekker, 1985.
14. G. Kelly and J. Power. Adjunctions whose counits are coequalizers and presentations of enriched monads. *J.Pure Appl. Algebra*, 89, 1993.
15. M. Kracht. *Tools and Techniques in Modal Logic*. Elsevier, 1999.
16. C. Kupke, A. Kurz, and D. Pattinson. Algebraic semantics for coalgebraic logics. In *CMCS'04*, ENTCS, 2004.
17. C. Kupke, A. Kurz, and D. Pattinson. Ultrafilter extensions of coalgebras. In *CALCO 2005*, LNCS 3629, 2005.
18. C. Kupke, A. Kurz, and Y. Venema. Stone coalgebras. *Theoret. Comput. Sci.*, 327, 2004.
19. A. Kurz. Specifying coalgebras with modal logic. *Theoret. Comput. Sci.*, 260, 2001.
20. F. Linton. An outline of functorial semantics. In *Seminar on triples and categorical homology theory*, LNM 80. 1969.
21. L. Moss. Coalgebraic logic. *Annals of Pure and Applied Logic*, 96, 1999.
22. L. Moss and I. Viglizzo. Harsanyi type spaces and final coalgebras constructed from satisfied theories. In *CMCS'04*, ENTCS, 2004.
23. D. Pattinson. Coalgebraic modal logic: Soundness, completeness and decidability of local consequence. *Theoret. Comput. Sci.*, 309, 2003.
24. J. Reiterman. Algebraic theories and varieties of functor algebras. *Fund. Math.*, 118, 1983.
25. E. Robinson. Powerdomains, modalities and the Vietoris monad. Technical report, Computer Laboratory Technical Report 98, University of Cambridge, 1986.
26. J. Rosický. On algebraic categories. In *Universal Algebra (Proc. Coll. Esztergom 1977)*, volume 29 of *Colloq. Math. Soc. J. Bolyai*, 1981.
27. M. Rößiger. Coalgebras and modal logic. In *CMCS'00*, volume 33 of *ENTCS*, 2000.
28. M. Rößiger. From modal logic to terminal coalgebras. *Theoret. Comput. Sci.*, 260, 2001.
29. J. Rutten. Universal coalgebra: A theory of systems. *Theoret. Comput. Sci.*, 249, 2000.
30. L. Schroeder. Expressivity of Coalgebraic Modal Logic: The Limits and Beyond. In V. Sassone, editor, *FoSSaCS'05*, volume 3441 of *LNCS*, 2005.
31. S. J. Vickers. *Topology Via Logic*. CUP, 1989.
32. G. Winskel. Note on powerdomains and modality. *Theoret. Comp. Sci.*, 36, 1985.

Bigraphical Models of Context-Aware Systems

L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, and H. Niss

IT University of Copenhagen (ITU)
{birkedal, debois, elsborg, hilde, hniss}@itu.dk

Abstract. As part of ongoing work on evaluating Milner's bigraphical reactive systems, we investigate bigraphical models of *context-aware systems*, a facet of ubiquitous computing. We find that naively encoding such systems in bigraphs is somewhat awkward; and we propose a more sophisticated modeling technique, introducing *Plato-graphical models*, alleviating this awkwardness. We argue that such models are useful for simulation and point out that for reasoning about such bigraphical models, the bisimilarity inherent to bigraphical reactive systems is not enough in itself; an equivalence between the bigraphical reactive systems themselves is also needed.

1 Introduction

The theory of *bigraphical reactive systems*, due to Milner and co-workers, is based on a graphical model of mobile computation that emphasizes both locality and connectivity [15, 19, 21]. A bigraph comprises a place graph, representing locations of computational nodes, and a link graph, representing interconnection of these nodes. We give dynamics to bigraphs by defining reaction rules that rewrite bigraphs to bigraphs; roughly, a bigraphical reactive system (BRS) is a set of such rules. Based on methods of the seminal [16], any BRS has a labelled transition system, the behavioural equivalence (bisimilarity) of which is a congruence.

There are two principal aims for the theory of bigraphical reactive systems: (1) to model ubiquitous systems [28], capturing mobile locality in the place graph and mobile connectivity in the link graph; and (2) to be a meta-theory encompassing existing calculi for concurrency and mobility. To date, the theory has been evaluated only wrt. the second aim: We have bigraphical understanding of Petri nets [18], π-calculus [13, 15, 14], CCS [21], mobile ambients [13], HOMER [5], and λ-calculus [19, 20].

The present paper initiates the evaluation of the first aim. We investigate modeling of *context-aware systems*, a vital aspect of ubiquitous systems. A context-aware application is an application that adapts its behaviour depending on the context at hand [26], interpreting "context" to mean the situation in which the computation takes place [10]. The canonical example of such a situation is the location of the device performing the computation; systems sensitive to location are called *location-aware*. As an example, a location-aware printing system could

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 187–201, 2006.
© Springer-Verlag Berlin Heidelberg 2006

send a user's print job to a printer close by. (For notions of context different from location, refer to [27]; for large-scale practical examples, see [1].)

To observe changes in the context, context-aware systems typically include a separate context sensing component that maintains a model of the current context. Such models are known as context models [12] or, more specifically, location models [2]. The above-mentioned location-aware printing system would need to maintain a model of the context that supports finding the printer closest to a given device. Such models are informal. There are only very few formal models of context-aware computing (refer to [11] for an overview). We point out Context Unity [25]; in spirit, our proposal is somewhat closer to process calculi than Context Unity is. However, bigraphs differ from traditional process calculi in that we get to write our own reaction rules.

In overall terms, our contribution is two-fold.

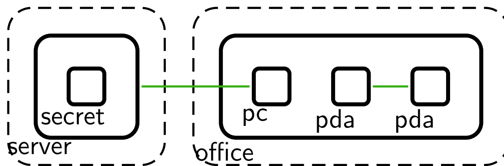
- We find, perhaps surprisingly, that naively modeling context-aware systems as BRSs is somewhat awkward; and
- we propose a more sophisticated modeling technique, in which the perceived and actual context are both explicitly represented as distinct but overlapping BRSs. We call such models *Plato-graphical*.

The remainder of this paper is organized as follows. In Section 2, we introduce bigraphs and bigraphical reactive systems. In Section 3, we discuss naive bigraphical models of location-aware systems. In Section 4, we introduce our Plato-graphical models of context-aware systems. In Section 5, we present two example models. In Section 6, we discuss. Finally, in Section 7, we conclude and note future work.

2 Bigraphs and Bigraphical Reactive Systems

We introduce bigraphs by example (the reader can find the relevant formal definitions of [15, 21] in Appendix A of [3]). Readers acquainted with bigraphs may skip this section.

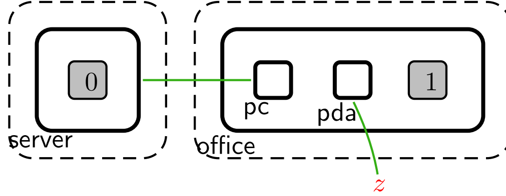
Here is a bigraph, A:



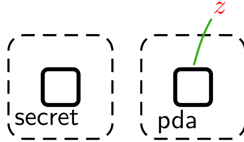
It has *nodes* (vertices), indicated by solid boxes. Each node has a *control*, written in sans serif. Each control has a number of *ports*; ports can be linked by *edges*, indicated by lines. Here, the controls secret and office have no ports, all other controls have one port. Nodes can be nested, indicated by containment. The two outermost dashed boxes indicate *roots*. Roots have no controls; they serve solely to separate different nesting hierarchies.

The bigraph A ostensibly models two physically separate locations (because of the two roots). The first contains a server, which in turn contains secret data; the second contains an office, which in turn contains a PC and two PDAs. The server and the PC are connected, as are the PDAs.

Here is another bigraph, B:



B resembles A, except that the content of server has been replaced with a *site* $-_0$, one of the pdas has been replaced by a site $-_1$, and there is an *inner name* z connected to the remaining pda. Using sites and names, we can define composition of bigraphs. For that, here is yet another bigraph C:



C has an *outer name* z. The bigraphs B and C compose to form A, i.e., $A = B \circ C$. Composition proceeds by plugging the roots of C into the sites of B (in order), and fusing together the connections $\mathsf{pda} \rightarrow z$ (in C) and $z \rightarrow \mathsf{pda}$ (in B) removing the name z in the process.

One cannot compose arbitrary bigraphs. For $U \circ V$ to be defined, U must have exactly as many sites as V has roots, and the inner names of U must be precisely the outer names of V. The sites and inner names are collectively called the *inner face*; similarly, the roots and outer names are called the *outer face*. A has inner face $\langle 0, \emptyset \rangle$ (no holes, no inner names) and outer face $\langle 2, \emptyset \rangle$ (two roots, no outer names). We write $A : \langle 0, \emptyset \rangle \rightarrow \langle 2, \emptyset \rangle$. Similarly, $B : \langle 2, \{z\} \rangle \rightarrow \langle 2, \emptyset \rangle$ and $C : \langle 0, \emptyset \rangle \rightarrow \langle 2, \{z\} \rangle$.

The graphical representation used above is handy for modeling, but unwieldy for reasoning. Fortunately, bigraphs have an associated term language [7, 17], which we use (albeit in a sugared form) in what follows. The language is summarized in Table 1. Here are, in order of increasing complexity, term representations of the bigraphs A, B and C.

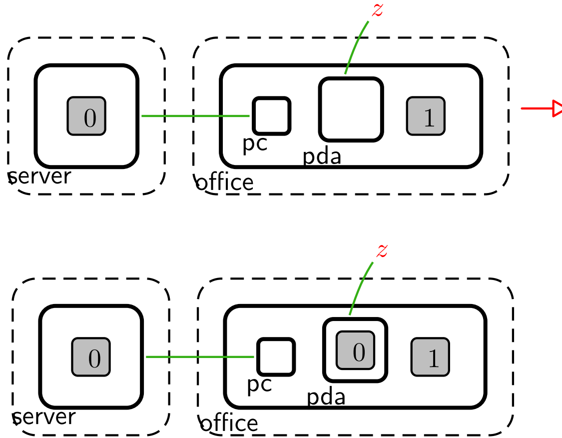
$$C = \mathsf{secret} \parallel \mathsf{pda}_z$$
$$A = /x./y.\mathsf{server}_x(\mathsf{secret}) \parallel \mathsf{office}(\mathsf{pc}_x \mid \mathsf{pda}_y \mid \mathsf{pda}_y)$$
$$B = /x./y.\mathsf{server}_x(-_0) \parallel \mathsf{office}(\mathsf{pc}_x \mid \mathsf{pda}_y \mid -_1) \mid y/z$$

Notice how, in B, edges are specified by first linking nodes to the same name, then converting that name to an edge using the closure '/'.

Table 1. Sugared term language for bigraphs

Term	Meaning
$U \parallel V$	Concatenation (juxtaposition) of roots.
$U \mid V$	Concatenation (juxtaposition) of children. (collect the children of U and V under one root.)
$U \circ V$	Composition.
$U(V)$	Nesting. U contains V.
$\mathsf{K}_{\vec{x}}(U)$	Ion. Node with control K of arity $\lvert\vec{x}\rvert$, ports connected to the outer names of vector $\vec{x}$. The node contains U.
1	The *barren* (empty) root.
$-_i$	Site numbered i.
$/x.U$	U with outer name x replaced by an edge.
x/y	Connection from inner name y to outer name x.

We give dynamics to bigraphs by defining reaction rules. Example:



$$/x.\mathsf{server}_x(-_0) \parallel \mathsf{office}(\mathsf{pc}_x \mid \mathsf{pda}_z \mid -_1)$$
$$\longrightarrow /x.\mathsf{server}_x(-_0) \parallel \mathsf{office}(\mathsf{pc}_x \mid \mathsf{pda}_z(-_0) \mid -_1)$$

This rule might model that if a PC in some office is linked to a server, a PDA in the same office may use the PC as a gateway to copy data from the server. The rule matches the bigraph A above, taking secret to the site $-_0$ and pda_y to the site $-_1$, rewriting A to

$$A' = /x./y.\mathsf{server}_x(\mathsf{secret}) \parallel \mathsf{office}(\mathsf{pc}_x \mid \mathsf{pda}_y(\mathsf{secret}) \mid \mathsf{pda}_y)$$

(We omit details on what it means to match connections; refer to one of [15, 21].)

It is occasionally convenient to limit the contexts in which a reaction rule applies [4], i.e., we might want to limit the above example reaction rule to apply only in the left wing of the building. To this end, bigraphs can be equipped

with a *sorting* [13, 21, 18]. A sorting consists of a set of *sorts* (or types); all inner and outer faces are then enriched with such a sort. Further, a sorting must stipulate some condition on bigraphs, we then restrict our attention to the bigraphs that satisfy that condition, thus outlawing some contexts. Obviously, removing contexts may ruin the congruence property of the induced bisimilarity; [13] and [21] give different sufficient conditions for a sorting to preserve that congruence property.

This concludes our informal overview of bigraphs. Now on to the models.

3 Naive Models of Location-Aware Systems

In this section, we attempt to model location-aware systems naively in bigraphs. We will find the naive approach to be somewhat awkward. Due to space constraints we do not discuss other forms of context.

We use the place and link graphs for describing locations and interconnections directly, and we use reaction rules to implement both *reconfiguration* of the context and *queries* on the context. The former is simply a non-deterministic change in the context; the latter is a computation on the context that does not change the context, except for producing an answer to some question. In a location-aware system, a device moving would be a reconfiguration, whereas computing the answer to the question "what devices are currently at the location l" is a query.

We discuss the implementation of this query. (An implementation of the query can be found in Appendix B in [3].) Incidentally, a query such as "find nearest neighbor", which conceptually is only slightly harder, is significantly harder to implement. (Other examples plagued by essentially the same difficulties can be found in [9].)

Consider the following bigraph representing devices (e.g., PDAs) located at locations (e.g., offices, meeting rooms) within a building.

$$l = /w./x./y./z.\mathsf{loc}\,(\mathsf{loc}\,(\mathsf{loc}\,(\mathsf{loc}\,(\mathsf{dev}_w) \mid \mathsf{loc}\,(\mathsf{dev}_x \mid \mathsf{dev}_y))) \mid \mathsf{loc}() \mid \mathsf{loc}\,(\mathsf{dev}_z))$$

Off-hand, finding all devices, say, beneath the root, looks straightforward: We should simply recursively traverse the nesting tree. Unfortunately, such traversal is quite complicated for the following reasons.

- The bigraphical reaction rules do not support recursion directly, so we must encode a runtime stack by means of additional controls.
- Bigraphical reaction rules can be applied in *any* context, but when implementing an operation such as the query we consider now, we need more refined control over when rules can be applied; one may achieve this more refined control by again using additional nodes and controls, essentially implementing what corresponds to a program counter. This still leaves great difficulty in handling concurrent operations, though.
- As a special case of the previous item, it is particularly difficult to express that a reaction rule is intended to apply only in case something is *not* present in the context.

Summing up, the bigraphical rules that model physical action do not in general provide the power to compute directly with a model of that action (because of a lack of control structures). The slogan is "reconfiguring is easy, querying is hard".

In earlier work on evaluating bigraphs as a meta-theory (aim (2) mentioned in the Introduction), reaction rules were used to encode the operational semantics of a calculus or programming language. However, above we attempt to implement a query *directly* as reaction rules. This seemingly innocuous difference will turn out to have major implications for reasoning methods; more on this in Section 6.

We imagine that adding more flexibility to the reaction rules might make it easier to program directly with bigraphs. One possible attempt is to use spatial logics for bigraphs [6] in combination with sorting, to get control of the contexts in which a particular reaction rule applies.

In the following sections, we propose another way to model context-aware systems in bigraphs, where the reaction rules are not used to program directly with but instead they are used (1) to represent transitions happening in the real world and (2) to encode operational semantics of programming languages, within which one can then implement queries on representations of the real world.

4 Plato-Graphical Models of Context-Aware Systems

The naive model of the previous section shares an important characteristic with recent proposals of formal models for context-aware computation [4, 8, 25] that comprise agents and contexts only: These models take the agent's ability to determine *what is* the present context as given. We contend that for some systems, it is natural to model not only the actual context but also the agent's representation of the actual context. We shall see that pursuing this idea will partially alleviate the awkwardness seen in the previous Section.

We shall need some notation and definitions.

Notation 1. We write $\mathbf{B} = (\mathcal{K}, \mathcal{R})$ to indicate that $\mathbf{B}$ is a bigraphical reactive system with controls $\mathcal{K}$ and rules $\mathcal{R}$, and write $f \in \mathbf{B}$ to mean that f is a bigraph of $\mathbf{B}$.

Definition 1 (Independence). Let $\mathbf{B} = (\mathcal{K}, \mathcal{R})$ and $\mathbf{B}' = (\mathcal{K}', \mathcal{R}')$ be bigraphical reactive systems. Say that $\mathbf{B}$ and $\mathbf{B}'$ are *independent* and write $\mathbf{B} \perp \mathbf{B}'$ iff $\mathcal{K}$ and $\mathcal{K}'$ are disjoint.

Definition 2 (Composite bigraphical reactive systems). Let $\mathbf{B} = (\mathcal{K}, \mathcal{R})$ and $\mathbf{B}' = (\mathcal{K}', \mathcal{R}')$ be bigraphical reactive systems. Define the union $\mathbf{B} \cup \mathbf{B}'$ point-wise, i.e., $\mathbf{B} \cup \mathbf{B}' = (\mathcal{K} \cup \mathcal{K}', \mathcal{R} \cup \mathcal{R}')$, when $\mathcal{K}$ and $\mathcal{K}'$ agree on the arities of the controls in $\mathcal{K} \cap \mathcal{K}'$.

Be aware that there are two ways of taking the union of two sets of *parametrized* reaction rules: (1) merge the rules and then ground them, or (2) first ground the rules and then merge them. In general, the resulting rule set of (1) is a superset of the rule set of (2). We use approach (1).

We propose a model of context-aware computing that comprises three bigraphical reactive systems: the context $\mathbf{C}$; its representation or proxy $\mathbf{P}$; and the computational agents $\mathbf{A}$. Drawing on classical work [23], specifically The Allegory of the Cave, we call such a model *Plato-graphical*.

Definition 3 (Plato-graphical model). A *Plato-graphical model* is a triple $(\mathbf{C}, \mathbf{P}, \mathbf{A})$ of bigraphical reactive systems, such that $\mathcal{M} = \mathbf{C} \cup \mathbf{P} \cup \mathbf{A}$ is itself a bigraphical reactive system and $\mathbf{C} \perp \mathbf{A}$. A *state* of the model is a bigraph of $\mathcal{M}$ on the form $/\vec{x}.(C \parallel P \parallel A)$, where $C \in \mathbf{C}$, $P \in \mathbf{P}$, $A \in \mathbf{A}$, and $\vec{x}$ is some vector of names.

We emphasize the intended difference between $\mathbf{C}$ and $\mathbf{P}$: Whereas an element of $\mathbf{C}$ models a possible context, an element of $\mathbf{P}$ models *a model* of a possible context. The independence condition ensures that agents can only directly observe or manipulate the proxy; not the context itself. (In the parlance of [25], the independence condition ensures separability.) To query or alter the context, agents must use the proxy as a sensor and actuator.

Using bigraphs as our basic formalism gives us two things. First, we can write our own reaction rules. We claim that because of this ability, models become remarkably straightforward and intuitive; hopefully, the reader will agree after seeing our example models in the next section. Second, we automatically get a bisimilarity that is a congruence. Thus, bisimilarity of agents is a very fine equivalence: No state of the context and proxy can distinguish bisimilar agents.

Proposition 1. *Let $\sim$ denote the bisimilarity in $\mathcal{M}$, and let $A, A' \in \mathbf{A}$ with $A \sim A'$. For any $C \in \mathbf{C}$, $P \in \mathbf{P}$, and $\vec{x}$, we have $/\vec{x}.(C \parallel P \parallel A) \sim /\vec{x}.(C \parallel P \parallel A')$.*

To get a less discriminating equivalence we can consider agents under a particular state of the context, or a particular state of the system.

Definition 4. Let $\sim$ denote the bisimilarity in $\mathcal{M}$, and let $A, A' \in \mathbf{A}$, $C \in \mathbf{C}$ and $P \in \mathbf{P}$. We say A and A' are *equivalent w.r.t.* P iff $P \parallel A \sim P \parallel A'$, and we say A and A' are equivalent w.r.t. C, P iff $C \parallel P \parallel A \sim C \parallel P \parallel A'$.

We conjecture that the above forms of derived equivalences will prove useful for reasoning about a given Plato-graphical system.

Working within the Plato-graphical model, we are free to emphasize any of its three components, perhaps modeling $\mathbf{P}$ in great detail, but keeping $\mathbf{C}$ and $\mathbf{A}$ abstract.

Definition 3 above does not impose any restriction on composition of states. For example, assume that we have a Plato-graphical model $\mathcal{M} = (\mathbf{C}, \mathbf{P}, \mathbf{A})$, that c, p and a are controls of $\mathbf{C}$, $\mathbf{P}$ and $\mathbf{A}$, respectively, and that p is *not* a control of $\mathbf{C}$. Then the bigraphs

$$F = \mathsf{c}(-_0 \mid -_1) \parallel \mathsf{p} \parallel \mathsf{a}(-_2) \qquad \text{and} \qquad G = \mathsf{c} \parallel \mathsf{p} \parallel \mathsf{a}$$

are both states of $\mathcal{M}$, but their composite $F \circ G = \mathsf{c}(\mathsf{c} \mid \mathsf{p}) \parallel \mathsf{p} \parallel \mathsf{a}(\mathsf{a})$ is not a state of $\mathcal{M}$. This example implies that bisimilarity of states of a Plato-graphical system may be too fine a relation: Conceivably, when comparing two states s

and s', we may wish to take into account only contexts C such that $C \circ s$ and $C \circ s'$ are themselves states, i.e., we might want to outlaw F as a possible context for G. We can achieve this finer control using place-sorting. So, we define a place-sorted Plato-graphical model. The intuition behind our sorting is that we want to keep controls of $\mathbf{C}$, $\mathbf{P}$ and $\mathbf{A}$ separate when composing contexts of form $\mathbf{C} \parallel \mathbf{P} \parallel \mathbf{A}$.

Notation 2. Denote by $S_{i \leq m}$ a vector $m_0, \ldots, m_{n-1}$ of sorts. We will write $S_{i \leq m}$ for a sorted interface $\langle m, X, S_{i \leq m} \rangle$ when we do not care about names.

Definition 5 (Sorted Plato-graphical model). Let $\mathcal{M} = \mathbf{C} \cup \mathbf{P} \cup \mathbf{A}$ be a Plato-graphical model with $\mathbf{C} = (\mathcal{K}_\mathbf{C}, \mathcal{R}_\mathbf{C})$, $\mathbf{P} = (\mathcal{K}_\mathbf{P}, \mathcal{R}_\mathbf{P})$ and $\mathbf{A} = (\mathcal{K}_\mathbf{A}, \mathcal{R}_\mathbf{A})$. Define a sorting discipline on $\mathcal{M}$ by taking sorts $\Theta = \{\mathcal{K}_\mathbf{C}, \mathcal{K}_\mathbf{P}, \mathcal{K}_\mathbf{A}\}$ and, for primes, sorting condition $\Phi(f : S_{i \leq n} \to S) = \mathit{ctrl}(f) \subseteq S \wedge \forall i \leq n.\ S_i = S$, lifting to an arbitrary bigraph f' by decomposing f into primes $f' = f_0 \ldots f_{n-1}$ and declaring f' well-sorted iff all the f_i are. Let ϕ be an assignment of Θ-sorts to the rules of $\mathcal{R}_\mathbf{C}$, $\mathcal{R}_\mathbf{P}$, and $\mathcal{R}_\mathbf{A}$, such that every rule is well-sorted under Φ. Define $\mathcal{M}'$ to be $\mathcal{M}$ sorted by (Θ, Φ) (using ϕ to lift the reaction rules). In this case, we call $\mathcal{M}'$ a *sorted Plato-graphical model*, and define the states of $\mathcal{M}'$ to be the well-sorted bigraphs with outer face $\mathcal{K}_\mathbf{C}, \mathcal{K}_\mathbf{P}, \mathcal{K}_\mathbf{A}$.

The condition Φ essentially requires that (1) the controls of a prime (bigraph) are elements of the sort of its outer face, and (2) the sort of the outer face is exactly the sort of each of the sites. Under this sorting discipline and new definition of state, if G is assigned a sort such that it is a state, then F cannot be assigned a sort that makes it composible with G.

Is the bisimilarity in the sorted system $\mathcal{M}'$ a congruence? The sorting discipline of $\mathcal{M}'$ is in general not homomorphic in the sense of Milner [21, Definition 10.4]: we cannot give a sort to controls in $\mathcal{K}_\mathbf{C} \cap \mathcal{K}_\mathbf{P}$. (If $\mathbf{C}$, $\mathbf{P}$ and $\mathbf{A}$ are pairwise independent, the sorting is homomorphic; however, such a model is pathologic.) Neither is the sorting safe in the sense of Jensen [13, Definition 4.30]; condition (4) cannot be met. Counterexample: Suppose $f : \mathcal{K}_\mathbf{C} \to \mathcal{K}_\mathbf{C}$ is well-sorted; take $g = f \otimes 1 : \mathcal{K}_\mathbf{C} \to \mathcal{K}_\mathbf{C}, \mathcal{K}_\mathbf{A}$ (recall that $1 : \epsilon \to \langle 1, \emptyset \rangle$ denotes the barren root). Clearly, $\mathcal{U}(f) = (-_0 \mid -_1) \circ \mathcal{U}(f \otimes 1)$. However, if $\mathcal{K}_\mathbf{C} \neq \mathcal{K}_\mathbf{A}$ then $(-_0 \mid -_1) : \mathcal{K}_\mathbf{C}, \mathcal{K}_\mathbf{A} \to \mathcal{K}_\mathbf{C}$ is not well-sorted.

Nevertheless, the sorting of Definition 5 does give rise to a bisimilarity that is a congruence; we prove so in Appendix C in [3].

5 Examples

5.1 A Simple Context-Aware Printing System

We model the simple context-aware printing system of [4]. An office-building contains both modern PCL-5e compatible printers and old-fashioned raw-printers. Occasionally, the IT-staff at the building removes or replaces either type of printers. Each printer can process only one job; queueing is done by a central print server. The print server dispatches jobs to raw-printers only if it knows no PCL-printers; if there are PCL-printers, but they are all busy, the job will

simply have to wait. This system is context-aware: The type and number of printers physically available determine the meaning of the action “to print”. We give a model **B** of this system in Figure 1. Looking at the controls of **B**, it is straightforward to verify that **B** is Plato-graphical.

Proposition 2. *The model* **B** *of Figure 1 is Plato-graphical.*

We take a detailed look at the model. A state of the context **C** consists of nested physical locations loc, within which printers prt are placed. We distinguish between PCL- and raw-printers by putting a token pcl and raw within them, respectively. Each printer has a single port, intended to link the printer to the proxy. Here is a state of the context with a PCL-printer and a raw-printer at adjacent locations; the PCL-printer is idle whereas the raw-printer is busy.

$$C = \mathsf{loc}(\mathsf{loc}(\mathsf{prt}_x(\mathsf{raw} \mid \mathsf{dat}_z)) \mid \mathsf{loc}(/y.\mathsf{prt}_y(\mathsf{pcl})))$$

Setting C in parallel with some proxy P will allow P access to the raw printer through the shared link x, but not to the PCL-printer, because it is in a closed link. The dynamics of **C** allow printers to appear (1, 2), disappear (3), and finish printing (4).

A state of the proxy **P** consists of a pool of pending jobs jobs and two tables of printers prts; one contains a token raw, the other a token pcl, indicating what type of printer the table lists. The prts is a table in the sense that its only port is linked to all the printers in the context that the table knows about. Here is an example state of the proxy which knows one raw-printer, knows no PCL-printers and has two pending jobs.

$$P = \mathsf{prts}_x(\mathsf{raw}) \mid /y.\mathsf{prts}_y(\mathsf{pcl}) \mid \mathsf{jobs}(/z.\mathsf{doc}_z \mid /z'.\mathsf{doc}_{z'})$$

Setting C and P above in parallel by $\|$, and closing the link x, we get a system $/x.C \parallel P$, where the table $\mathsf{prts}_x(\mathsf{raw})$ and the physical printer $\mathsf{prt}_x(\mathsf{raw} \mid \mathsf{dat})$ are linked. The dynamics of **P** state that if there is a job and a known, idle PCL-printer, the proxy may activate this printer (5); that if there is a job, no known PCL-printer, and an idle raw-printer, the context may activate that printer (6); and finally, that the proxy may discover a previously unknown printer (7, 8).

The dynamics of **A** allow the agents to spontaneously spool documents (9).

Notice how the two printing rules (5) and (6) do not observe the context directly. Instead, the proxy observes the context (rules (7) and (8)) and records its observations in the tables $\mathsf{prts}_x(\mathsf{raw})$ and $\mathsf{prts}_y(\mathsf{pcl})$; the printing rules (5) and (6) then consults the tables. It is straightforward to determine whether there are no known PCL-printers: simply check if the table of PCL-printers has the form $/y.\mathsf{prts}_y(\mathsf{pcl})$.

As observed in Section 3 and [4], it is generally very difficult, if not impossible, to observe the *absence* of something in the context directly. An interesting but rather natural consequence of the indirect observation is that it becomes asynchronous, i.e., it is possible that a PCL-printer exists but has not yet been observed.

Context **C**.

Control	Activity	Arity	Comment
loc	active	0	Nested location
prt	passive	1	Physical printer
pcl	atomic	0	Printer-type token
raw	atomic	0	Printer-type token
dat	atomic	1	Binary data for printer

$$\mathsf{loc}(-_0) \longrightarrow \mathsf{loc}(-_0 \mid /x.\mathsf{prt}_x(\mathsf{raw})) \tag{1}$$

$$\mathsf{loc}(-_0) \longrightarrow \mathsf{loc}(-_0 \mid /x.\mathsf{prt}_x(\mathsf{pcl})) \tag{2}$$

$$\mathsf{loc}(-_0 \mid \mathsf{prt}_x(-_1)) \longrightarrow \mathsf{loc}(-_0) \mid x/ \tag{3}$$

$$\mathsf{prt}_x(\mathsf{dat}_z \mid -_0) \longrightarrow \mathsf{prt}_x(-_0) \mid z/ \tag{4}$$

Proxy **P**.

Control	Activity	Arity	Comment
prt	passive	1	Physical printer
pcl	atomic	0	Printer-type token
raw	atomic	0	Printer-type token
dat	atomic	1	Binary data for printer
prts	passive	1	Known devices
jobs	passive	0	Pending documents
doc	atomic	1	Document

$$\begin{aligned}&\mathsf{jobs}(\mathsf{doc}_z \mid -_0) \parallel \mathsf{prts}_y(\mathsf{pcl}) \parallel \mathsf{prt}_y(\mathsf{pcl}) \longrightarrow \\ &\qquad \mathsf{jobs}(-_0) \parallel \mathsf{prts}_y(\mathsf{pcl}) \parallel \mathsf{prt}_y(\mathsf{pcl} \mid \mathsf{dat}_z)\end{aligned} \tag{5}$$

$$\begin{aligned}&\mathsf{jobs}(\mathsf{doc}_z \mid -_0) \parallel /x.\mathsf{prts}_x(\mathsf{pcl}) \mid \mathsf{prts}_y(\mathsf{raw}) \parallel \mathsf{prt}_y(\mathsf{raw}) \longrightarrow \\ &\qquad \mathsf{jobs}(-_0) \parallel /x.\mathsf{prts}_x(\mathsf{pcl}) \mid \mathsf{prts}_y(\mathsf{raw}) \parallel \mathsf{prt}_y(\mathsf{raw} \mid \mathsf{dat}_z)\end{aligned} \tag{6}$$

$$/x.\mathsf{prt}_x(\mathsf{pcl}) \parallel \mathsf{prts}_y(\mathsf{pcl}) \longrightarrow \mathsf{prt}_y(\mathsf{pcl}) \parallel \mathsf{prts}_y(\mathsf{pcl}) \tag{7}$$

$$/x.\mathsf{prt}_x(\mathsf{raw}) \parallel \mathsf{prts}_y(\mathsf{raw}) \longrightarrow \mathsf{prt}_y(\mathsf{raw}) \parallel \mathsf{prts}_y(\mathsf{raw}) \tag{8}$$

Agents **A**.

Control	Activity	Arity	Comment
jobs	passive	0	Pending documents
doc	atomic	1	Document

$$\mathsf{jobs}(-_0) \longrightarrow \mathsf{jobs}(-_0 \mid /z.\mathsf{doc}_z) \tag{9}$$

Fig. 1. Example Plato-graphical model **B**

Context **C**	Proxy **P**	Agent **A**
(1) : $\mathcal{K}_{\mathbf{C}}$	(5) : $\mathcal{K}_{\mathbf{A}}, \mathcal{K}_{\mathbf{P}}, \mathcal{K}_{\mathbf{C}}$	(9) : $\mathcal{K}_{\mathbf{A}}$
(2) : $\mathcal{K}_{\mathbf{C}}$	(6) : $\mathcal{K}_{\mathbf{A}}, \mathcal{K}_{\mathbf{P}}, \mathcal{K}_{\mathbf{C}}$	
(3) : $\mathcal{K}_{\mathbf{C}}$	(7) : $\mathcal{K}_{\mathbf{P}}, \mathcal{K}_{\mathbf{C}}$	
(4) : $\mathcal{K}_{\mathbf{C}}$	(8) : $\mathcal{K}_{\mathbf{P}}, \mathcal{K}_{\mathbf{C}}$	

Fig. 2. Sorts for the rules of **C**, **P**, and **A**

This model **B** can be lifted to a sorted one by adding the sorts given in Figure 2; the figure assigns sorts to the outer face of both the redexes and reactums of the indicated rules. It is straightforward to verify that all of the rules are well-sorted.

Proposition 3. *The model* **B** *with the sorting assignment of Figure 2 is a sorted Plato-graphical model.*

5.2 A Location-Aware Printing System

Suppose we extend the printing system with location-awareness, by stipulating that a print job is not printed until the printer and the device submitting the job are co-located. To model this extended system, we introduce a new control dev for devices (PCs or PDAs) with one port and change doc to include an extra port so we can link submitted jobs to the devices submitting them. The linking is reflected in the following modified rule (9) for spooling print jobs:

$$\mathsf{loc}(\mathsf{dev}_x \mid -_0) \parallel \mathsf{jobs}(-_1) \longrightarrow \mathsf{loc}(\mathsf{dev}_x \mid -_0) \parallel \mathsf{jobs}(-_1 \mid /z.\mathsf{doc}_{z,x}) \qquad (9')$$

We must also modify rules (5) and (6) to insist that the device and printer are co-located. Rule (5) becomes

$$\begin{aligned} &\mathsf{jobs}(\mathsf{doc}_{z,x} \mid -_0) \parallel \mathsf{prts}_y(\mathsf{pcl}) \parallel \mathsf{loc}(\mathsf{dev}_x \mid \mathsf{prt}_y(\mathsf{pcl})) \longrightarrow \\ &\qquad \mathsf{jobs}(-_0) \parallel \mathsf{prts}_y(\mathsf{pcl}) \parallel \mathsf{loc}(\mathsf{dev}_x \mid \mathsf{prt}_y(\mathsf{pcl} \mid \mathsf{dat}_z)). \end{aligned} \qquad (5')$$

(We suppress the new Rule (6').)

Modifying the system once again, instead of insisting that device and printer have to be actually co-located, we just require the print job to end at a printer close to the device. The print server will need to query the proxy for the printer nearest a given device. We saw in Section 3 that implementing such queries is awkward, so we will need to use the proxy. In the preceding Section, we did so directly in bigraphs; this time around, we transfer the expressive convenience of a general-purpose programming language to bigraphs for ease of implementation. We use bigraphs directly for modeling the actual context **C**, whereas we will exploit bigraphs as a meta-calculus for modeling the proxy **P**.

In detail, the whole model is $\mathbf{B} = \mathbf{C} \cup \mathbf{P} \cup \mathbf{A}$, with $\mathbf{P} = \mathbf{S} \cup \mathbf{L}$. Here **C** is intended to be a bigraphical model of the "real world", the proxy **P** is comprised of a location sensor **S** and a location model **L** and **A** is the location-based application (the "computational agent").

A state C of **C** could look like this:

$$C = \mathsf{loc}(\mathsf{loc}(\mathsf{loc}(\mathsf{loc}(\mathsf{dev}_w) \mid \mathsf{loc}(\mathsf{dev}_x \mid \mathsf{dev}_y))) \mid \mathsf{loc} \mid \mathsf{loc}(\mathsf{dev}_z))$$

Changes in the real world are modeled by reaction rules that reconfigure such states. If we want to model, say, that a devices may move from one location to another, we include the reaction rule

$$\mathsf{loc}(\mathsf{dev}_x \mid -_0) \parallel \mathsf{loc}(-_1) \longrightarrow \mathsf{loc}(-_0) \parallel \mathsf{loc}(\mathsf{dev}_x \mid -_1). \qquad (10)$$

To implement the proxy, encode as a BRS a programming language $\mathcal{L}$ with data structures, communication primitives, and concurrency, e.g., Pict [22] or CML [24]. (We return to this assumption below.) That is, define a translation from terms of $\mathcal{L}$ to bigraphs, and add reaction rules encoding the operational semantics of $\mathcal{L}$. *Then implement the location model, the sensor, and the agents in $\mathcal{L}$ and use the encoding to transfer that model to bigraphs.* In particular, a state of the location model **L** will have a data structure *representing* the current state of **C**. If $\mathcal{L}$ is an even half-way decent programming language, it should be straightforward to implement queries such as one of Section 3 or the "find closest printer" we need above.

The sensor informs the location model about changes in **C**. We extend the above rule (10) moving a device to

$$(\mathsf{loc}(\mathsf{dev}_x \mid -_0) \parallel \mathsf{loc}(-_1)) \mid S \mid L \longrightarrow (\mathsf{loc}(-_0) \parallel \mathsf{loc}(\mathsf{dev}_x \mid -_1)) \mid S' \mid L, \quad (10')$$

where S' is an $\mathcal{L}$-encoding of "send a notification to **L** that device x has moved". Upon receiving the notification, **L** updates its representation of the world. Agents of **A** can in turn query **L** when they need location information.

6 Discussion

We consider the following questions.

1. What languages $\mathcal{L}$ can we encode?
2. How close are Plato-graphical models to real systems?
3. What challenges have we found for bigraphical models?
4. What uses do we envision for Plato-graphical models?
5. How do we reason about Plato-graphical models?

Ad. 1. As mentioned, there exist bigraphical encodings of various π-calculi [13, 15, 14] and of the λ-calculus [19, 20]. Using ideas of the latter encodings, we have encoded Mini-ML (call-by-value λ-calculus with pairs and lists) in local bigraphs [19]. Based on our experiences with this encoding, we find it palatable to encode CML or Pict[1].

Ad. 2. The model closely reflects how some actual location-aware systems work, for instance the one running at the ITU. Here, a sensor system (made by Ekahau) computes every two seconds the physical location of every device on the WLAN. The sensor system informs a location model about updates to locations; location-aware services then interact with the location model. In our sketched Plato-graphical model, the location model **L** may lag behind the actual **C**, if **L**'s representation of **C** does not reflect some recent reconfiguration of **C**. But that also happens in the real system at the ITU – when a location-aware service asks the location model for the whereabouts of a device, it obtains not the position

[1] We are presently working on implementing an interpreter for bigraphical reactive systems; such an interpreter will make it easier to experiment with these and other encodings.

of the device, but the position of the device the last time the sensor checked. In the mean time, the device may have moved.

Ad. 3. When modeling the physical world, we have made use of both the place and link graphs, the place graph modeling the location hierarchy of a building. As argued in [2], DAGs or graphs are more natural models of location. Thus, systems such as the ones we have considered here suggest generalizing the place graphs part of bigraphs, or consider ways to encode DAGs or general graphs naturally as place graphs.

Ad. 4. Given an implementation of bigraphical reactive systems, one could *simulate* the behaviour of a location-aware system, and thus allow for experimentation with different designs of location-aware and context-aware systems. Likewise, one could experiment with different choices for the $\mathcal{L}$ language of Section 5.2. Such simulation suggests further extensions of the bigraphical model: In actual context-aware systems, one is generally interested in timing aspects (e.g., the sensor samples only every two seconds), continuous space (e.g., the sensor really produces continuous data), and probabilistic models (e.g., to accurately simulate sensors and sensor failure).

Ad. 5. What about using Plato-graphical models for *formal reasoning* about context-aware systems? One use of formal models is to prove an abstract specification model equivalent to a concrete implementation model. In π-calculus, we come with π-terms i, s, one for the implementation and one for the specification. The terms i and s are themselves the models; we take (π-) bisimilarity as equivalence, so to prove i and s equivalent, we merely prove them bisimilar. We can play the same game within any BRS: Simply come up with a bigraph I (the implementation model) and a bigraph S (the specification model), and prove them bisimilar within the labelled transition system of the BRS. Because that bisimulation is a congruence, such reasoning should be tractable, e.g. with the bisimulation in Definition 4.

Unfortunately, bisimulation within a single BRS is not always enough wrt. Plato-graphical models. Suppose we want a specification model $\mathcal{M}$ with an abstract view of the context, and an implementation model $\mathcal{M}'$ with a detailed view of the context. We express this by having $\mathcal{M}$ and $\mathcal{M}'$ differ only in their context sub-BRSs, that is,

$$\mathcal{M} = \mathbf{C} \cup \mathbf{P} \cup \mathbf{A} \qquad \mathcal{M}' = \mathbf{C}' \cup \mathbf{P} \cup \mathbf{A}.$$

The trouble is that because $\mathbf{C}$ and $\mathbf{C}'$ may have different controls and reaction rules, bisimulation between their respective labelled transition systems is meaningless! What we need is a notion of equivalence of BRSs, not just equivalence of bigraphs of a single BRS. At the time of writing, we know of no such equivalence[2]. Thus, our investigation of bigraphical models for context-aware systems

[2] The reader may suggest that we just define a common language for modeling both the abstract and detailed view, and define a translation from this language into a single BRS. However, in this case we are no longer modeling a ubiquitous system directly in bigraphs (aim 1 of the Introduction), but using bigraphs as a meta-calculus (aim 2 of the Introduction).

suggests that equivalence of BRSs is a key notion currently missing. One possible direction would be to try recover from the notion of WRS-functor [16] — functors that preserve reaction rules — a notion of a BRS implementing another BRS.

7 Conclusion and Future Work

We have initiated an evaluation of the use of bigraphical reactive systems for models of context-aware computing in ubiquitous systems. We found that BRSs, in their current form, are not suitable for directly modeling context queries, but on the other hand suitable for modeling reconfigurations of the actual context.

In response, we proposed Plato-graphical models, where both state and dynamics are logically divided in three parts: the actual context, the observed context (or proxy), and the computational agents, respectively. The computational agents and the actual context are separated, and interact only through the proxy. This separation into different BRSs makes it possible to encode different parts of the system using domain-specific languages. Moreover, we have shown how the context-aware printing system of [4] can be modeled straightforwardly in the Plato-graphical model.

Further, we have argued that Plato-graphical models are useful for simulating context-aware systems, and we are currently working on an implementation of BRSs at ITU to allow such experimentation. Only through such experimentation will it be clear how useful Plato-graphical models really are. For simulation purposes it will be important to extend bigraphs with timing aspects, continuous space, and probabilities.

Finally, we have pointed out that establishing a notion of equivalence between BRSs, as opposed to bisimilarity within a BRS, is important future work.

Acknowledgments

We gratefully acknowledge discussions with the other members of the BPL group at ITU, in particular Arne Glenstrup, Troels Damgaard and Mikkel Bundgaard; and with Robin Milner. This work was funded in part by the Danish Research Agency (grant no.: 2059-03-0031) and the IT University of Copenhagen (the LaCoMoCo project).

References

1. M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a sentient computing system. *IEEE Computer*, 2001.
2. C. Becker and F. Dürr. On location models for ubiquitous computing. *Personal and Ubiquitous Computing*, 9:20–31, 2005. Springer.
3. L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, and H. Niss. Bigraphical Models of Context-aware Systems. Technical Report 74, IT Univ. of Copenhagen, 2005.

4. P. Braione and G. P. Picco. On calculi for context-aware coordination. In *Proc. of COORDINATION'04*, vol. 2949 of *LNCS*, pages 38–54. 2004.
5. M. Bundgaard and T. Hildebrandt. Bigraphical semantics of higher-order mobile embedded resources with local names. In *Proc. of GT-VC'05*, 2005.
6. G. Conforti, D. Macedonio, and V. Sassone. Spatial Logics for Bigraphs. In *Proc. of ICALP'05*, vol. 3580 of *LNCS*, pages 766–778. 2005.
7. T. C. Damgaard and L. Birkedal. Axiomatizing binding bigraphs (revised). Technical Report TR-2005-71, IT University of Copenhagen, 2005.
8. R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. In *Proc. of ICALP'05*, volume 3580 of *LNCS*, pages 1226–1238. Springer, 2005.
9. S. Debois and T. C. Damgaard. Bigraphs by Example. Technical Report TR-2005-61, IT University of Copenhagen, March 2005.
10. A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on The What, Who, Where, When, and How of Context-Awareness*, 2000.
11. M. Hennessy. Context-awareness: Models and analysis. Talk at 2nd UK-UbiNet Workshop, 2004.
12. K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling context information in pervasive computing systems. In *Proc. of Pervasive'02*, vol. 2414 of *LNCS*, 2002.
13. O. H Jensen. *Mobile Processes in Bigraphs.* PhD thesis, 2005. Forthcoming.
14. O. H. Jensen and R. Milner. Bigraphs and Transitions. In *Proc. of POPL'03*, 2003.
15. O. H. Jensen and R. Milner. Bigraphs and mobile processes (revised). Technical Report UCAM-CL-TR-580, University of Cambridge, 2004.
16. J. J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. In *Proc. of CONCUR'00*, 2000.
17. R. Milner. Axioms for bigraphical structure. Technical Report UCAM-CL-TR-581, University of Cambridge, 2004.
18. R. Milner. Bigraphs for Petri Nets. In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, vol. 3098 of *LNCS*, 2004.
19. R. Milner. Bigraphs whose names have multiple locality. Technical Report UCAM-CL-TR-603, 2004.
20. R. Milner. Bigraphs: A tutorial. Slides, April 2005. Available at `http://www.cl.cam.ac.uk/users/rm135/bigraphs-tutorial.pdf` .
21. Robin Milner. Pure bigraphs: Structure and dynamics. To appear in *Information and Computation*, 2005.
22. B. C. Pierce and D. N. Turner. Pict: A prog. lang. based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of R. Milner*, MIT, 2000.
23. Plato. The republic, book vii, 360 B.C. Translation by Benjamin Jowett.
24. J H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
25. G.-C. Roman, C. Julien, and J. Payton. A formal treatment of context-awareness. In *Proc. of FASE'04*, vol. 2984 of *LNCS*, 2004.
26. B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proc. of IEEE Workshop on Mobile Computing Systems and Applications*, 1994.
27. A. Schmidt, M. Beigl, and H.-W. Gellersen. There is more to context than location. *Computers & Graphics Journal*, 1999.
28. M. Weiser. Hot topics – ubiquitous computing. *IEEE Computer*, 1993.

Processes for Adhesive Rewriting Systems*

Paolo Baldan[1], Andrea Corradini[2], Tobias Heindel[3],
Barbara König[3], and Paweł Sobociński[4]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany
[4] Computer Laboratory, University of Cambridge, United Kingdom

Abstract. Rewriting systems over adhesive categories have been recently introduced as a general framework which encompasses several rewriting-based computational formalisms, including various modelling frameworks for concurrent and distributed systems. Here we begin the development of a truly concurrent semantics for adhesive rewriting systems by defining the fundamental notion of process, well-known from Petri nets and graph grammars. The main result of the paper shows that processes capture the notion of true concurrency—there is a one-to-one correspondence between concurrent derivations, where the sequential order of independent steps is immaterial, and (isomorphism classes of) processes. We see this contribution as a step towards a general theory of true concurrency which specialises to the various concrete constructions found in the literature.

1 Introduction

Many rewriting theories have been developed in order to describe rule-based transformations over specific classes of objects: words (formal languages), terms, multi-sets (Petri nets) and graphs (graph rewriting). The recently introduced categorical foundation for double-pushout (DPO) rewriting theory based on *adhesive categories* [13] encompasses rewriting on words, multi-sets and (typed) graphs. Indeed, adhesive categories satisfy practically all the High-Level Replacement conditions [8], which ensure the validity of several standard theorems.

As a consequence of the relatively simple axioms and closure properties of adhesive categories, it is not difficult to show that a wide range of structures form the objects of an adhesive category. For instance, the categories of graphs with second-order edges or graphs with scopes are adhesive. Because of their generality, adhesivity and related concepts have begun to be exploited in the area of graph transformation (see e.g., [9]).

The view of adhesive rewriting systems as a general, unifying setting into which several models of concurrent and distributed systems can be embedded, calls for a generalization of the concurrency theory already developed for specific

* Partially supported by EPSRC grant GR/T22049/01, DFG project SANDS, EC RTN 2-2001-00346 SegraVis, MIUR project PRIN 2005015824 ART and EU IST-2004-16004 SEnSOria.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 202–216, 2006.
© Springer-Verlag Berlin Heidelberg 2006

formalisms like Petri nets and graph rewriting to this framework. The first steps in this direction were already taken in [13] where the notions of sequential and parallel independence of two rewriting steps, i.e. conditions under which they can be switched or applied concurrently, were studied.

In this paper we continue the development of a truly concurrent semantics for adhesive rewriting systems by generalizing the fundamental notion of process, well-known from the theory of Petri nets [11]. A process describes a possible computation of a given rule-based system taking into account the dependencies between the rewriting steps. The fact that two events are concurrent is modeled by the absence of dependencies between them. Intuitively, a process provides a canonical representation of a class of *derivations* (sequences of rewriting steps) which differ only in the order of independent rewriting steps.

The theory of processes and their correspondence with suitable equivalence classes of derivations has been generalized from nets to graph transformation systems in [6, 4, 1]. These approaches rely on the set-theoretical concept of *items*—tokens in the case of Petri nets, nodes and edges in the case of graph rewriting. For example, a transition t is said to be a *cause* of another transition t' if it produces a token in the pre-set of t', while in DPO-graph rewriting a rule cannot be applied to a graph if it deletes a node without deleting all edges incident to it (the so-called *dangling condition*).

In the abstract setting of adhesive categories, a concept related to the notion of *item* is that of *subobject* of an object X. A subobject is an isomorphism class of monomorphisms into X. For example, in the category of sets and functions, the subobjects of a set are (in 1-1-correspondence with) its subsets, while in the category of graphs and homomorphisms, a subobject of a graph is a subgraph. When working with subobjects of an object X in adhesive categories, we benefit from the fact that they form a distributive lattice [13]. However, we have no notion of "atoms" that can be consumed or produced. As a consequence, the techniques involved in the development of our theory are significantly different from those used in the setting of nets or graph rewriting and an original approach is needed in order to deal with the relevant concepts such as causality, concurrency, and negative application conditions for rules.

From a theoretical perspective, the central merit of our development lies in readdressing in the abstract setting of adhesive categories the concept of process that has so far been defined only in concrete cases. This is in contrast to related notions such as parallel and sequential-independence which are traditionally defined at the abstract level. The advantages of understanding processes at a general level are clear: we are able to prove theorems without resorting to the use of low-level structure.

The theory in this paper provides the foundations for the development of partial order verification methods that are applicable to rewriting systems over general "graph-like" structures, including, for instance, UML models, bigraphs and dynamic heap-allocated pointer structures.

Structure of the paper. We recall the definition of adhesive categories as well as some of their properties in §2. Adhesive grammars and derivations are introduced

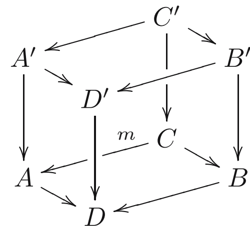
in §3 followed by a study of the possible relations among the rules and their connections with concurrency. The notion of occurrence grammars (on which the notion of process is based) is developed in §4. Finally, in §5 we define processes and show that processes and switch-equivalence classes of typed derivations are in 1-1-correspondence.

2 Adhesive Categories

Adhesive categories were introduced in [13]. Roughly, they may be described as categories where pushouts along monomorphisms are "well behaved". Here we only give a minimal introduction, concentrating on the algebra of subobjects of a given object T.

Definition 1 (Adhesive category). A category $\mathbf{C}$ is said to be *adhesive* if

1. $\mathbf{C}$ has pushouts along monomorphisms;
2. $\mathbf{C}$ has pullbacks;
3. Given a cube diagram as shown to the right with: (*i*) $m: C \to A$ mono, (*ii*) the bottom face a pushout and (*iii*) the back faces pullbacks, we have that the top face is a pushout iff the front faces are pullbacks.



The archetypal adhesive category is the category **Set** of sets and functions. Adhesive categories enjoy useful closure properties, for example if $\mathbf{C}$ is adhesive then so is any functor category $\mathbf{C}^{\mathbf{X}}$, any slice category $\mathbf{C} \downarrow C$ and any co-slice category $C \downarrow \mathbf{C}$. Therefore, since the category of graphs and graph morphisms is a functor category $\mathbf{Graph} \cong \mathbf{Set}^{\bullet \leftleftarrows \bullet}$, it is adhesive, and given a type graph T, the category of typed graphs $\mathbf{Graph} \downarrow T$ is adhesive.

A subobject of a given object T is an isomorphism class of monomorphisms to T. Binary intersections of subobjects exist in any category with pullbacks. Adhesive categories enjoy also the existence of binary subobject unions which are calculated in an intuitive way by pushing out along their intersection. Moreover, the lattice of subobjects is distributive.

Theorem 2 ([13], Theorem 17 and Corollary 18). *For an object T of an adhesive category $\mathbf{C}$, the poset* $\mathrm{Sub}(T)$ *of subobjects of T has joins: the join of two subobjects is (the isomorphism class of) their pushout in $\mathbf{C}$ over their intersection. Furthermore the lattice* $\mathrm{Sub}(T)$ *is distributive.*

3 Adhesive Grammars

We start by introducing rules and grammars. Rules consist of three objects: a left-hand side, a right-hand side and a common "read-only" part that is preserved, called the interface, which is a subobject of both the left- and the right-hand side.

Definition 3 (Rules and grammars). Let $\mathbf{C}$ be an adhesive category that we assume to be fixed for the rest of the paper. A *rule* is a span of monomorphisms $L \xleftarrow{\alpha} K \xrightarrow{\beta} R$ in $\mathbf{C}$. It is called *consuming* if α is not an isomorphism.

A *grammar* is a triple $\mathcal{G} = \langle S, P, \pi \rangle$, where P is a set of *rule names*, π is a function which maps any $q \in P$ to a rule $L_q \xleftarrow{\alpha_q} K_q \xrightarrow{\beta_q} R_q$ and $S \in \mathrm{ob}(\mathbf{C})$ is the *start object*. The grammar $\mathcal{G}$ is called consuming if all its rules are consuming.

A direct derivation is a diagram representing a single application of a rewriting rule. Applying several rules in sequence gives us a path through the state space of the grammar. The diagram consisting of the corresponding sequence of direct derivations can be reconstructed from a given path, and together they form a derivation.

Definition 4 (Direct derivations and paths). Let $\mathcal{G} = \langle S, P, \pi \rangle$ be a grammar, let $q \in P$, $A, B \in \mathrm{ob}(\mathbf{C})$, and $f \colon L_q \rightarrowtail A$ be a monomorphism. Then *q rewrites A to B at f in $\mathcal{G}$*, written $A \xRightarrow{\langle q,f \rangle}_{\mathcal{G}} B$, if there exists a diagram (1) consisting of two pushouts. If it exists, we shall refer to such a diagram as a *direct derivation along $\langle q, f \rangle$*, to D as *pushout complement* of α_q and f, and to f as a *(q-)match*.

$$\begin{array}{ccccc} L_q & \xleftarrow{\alpha_q} & K_q & \xrightarrow{\beta_q} & R_q \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle h} \\ A & \xleftarrow[\gamma]{} & D & \xrightarrow[\delta]{} & B \end{array} \tag{1}$$

A *$\mathcal{G}$-path* is a sequence $\tau = \langle q_i, f_i \rangle_{i \in [n]}$, so that $A_0 = S$ and $A_i \xRightarrow{\langle q_i, f_i \rangle}_{\mathcal{G}} A_{i+1}$ for $i \in [n]$.[1] Given a $\mathcal{G}$-path τ, let $\boldsymbol{d}^\tau$ be the diagram which results from including the direct derivations of all of τ's individual steps:

$$\begin{array}{ccccccccccccc} & L_0 \xleftarrow{\alpha_0} K_0 \xrightarrow{\beta_0} R_0 & & L_1 \xleftarrow{\alpha_1} K_1 \xrightarrow{\beta_1} R_1 & & \cdots & & L_{n-1} \xleftarrow{\alpha_{n-1}} K_{n-1} \xrightarrow{\beta_{n-1}} R_{n-1} \\ & f_0 \swarrow \quad \downarrow g_0 \quad \searrow h_0 & & f_1 \swarrow \quad \downarrow g_1 \quad \searrow h_1 & & \cdots & & f_{n-1} \swarrow \quad \downarrow g_{n-1} \quad \searrow h_{n-1} \\ S = A_0 & \xleftarrow[\gamma_0]{} D_0 \xrightarrow[\delta_0]{} & A_1 & \xleftarrow[\gamma_1]{} D_1 \xrightarrow[\delta_1]{} & A_2 & \cdots & A_{n-1} & \xleftarrow[\gamma_{n-1}]{} D_{n-1} \xrightarrow[\delta_{n-1}]{} & A_n \\ & \underbrace{\qquad\qquad\qquad}_{\boldsymbol{d}^\tau_0} & & \underbrace{\qquad\qquad\qquad}_{\boldsymbol{d}^\tau_1} & & & & \underbrace{\qquad\qquad\qquad}_{\boldsymbol{d}^\tau_{n-1}} \end{array}$$

Then $\boldsymbol{d}^\tau$ is said to be a *diagram of* τ and a *witness* of $A_0 \overset{\tau}{\Rightarrow} A_n$ and the pair $\langle \tau, \boldsymbol{d}^\tau \rangle$ is called a *$\mathcal{G}$-derivation*. For each $i \in [n]$ we write $\boldsymbol{d}^\tau_i$ for the sub-diagram of $\boldsymbol{d}^\tau$ that witnesses $A_i \xRightarrow{\langle q_i, f_i \rangle} A_{i+1}$, and $\boldsymbol{d}^\tau_{[i]}$ for the sub-diagram containing the first i steps of the derivation diagram. Each sub-diagram $L_i \xleftarrow{\alpha_i} K_i \xrightarrow{\beta_i} R_i$ is said to be an *occurrence of* q_i.

In the sequel we will consider typed grammars, as introduced in [6], which are grammars where every component is endowed with a morphism into a fixed

[1] For each $n \in \mathbb{N}$, we denote by $[n]$ the set $\{0, \ldots, n-1\}$.

object $T \in \mathrm{ob}(\mathbf{C})$. Roughly, the type object T is intended to provide the pattern which any possible system state must conform to, and the existence of the typing morphism $a\colon A \to T$ ensures that the state A conforms to the type.

Formally, typed grammars can be seen as grammars in the slice category $\mathbf{C}{\downarrow}T$, which is adhesive when $\mathbf{C}$ is (see [13]). However having an explicit typing will be useful when defining the process of a grammar $\mathcal{G}$, which describes a concurrent computation in $\mathcal{G}$ by representing the rules and the resources used in such a computation. Explicitly working with this type object will enable us to view all left-hand sides, right-hand sides and interfaces as subobjects and work in the subobject lattice $\mathrm{Sub}(T)$.

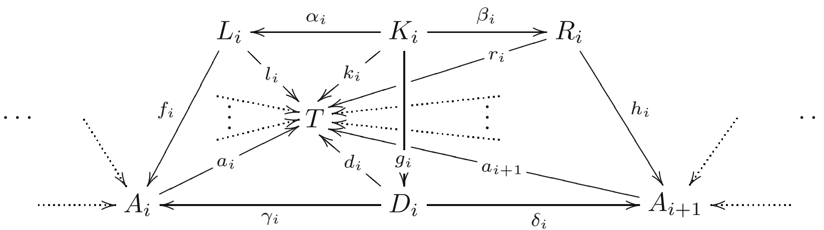
To describe the typed setting formally it shall be convenient to consider an "identity" rule for the start object of a grammar. Given $S \in \mathrm{ob}(\mathbf{C})$, we shall adopt the convention of letting $\underline{S}$ denote the rule $\pi(\underline{S}) = S \xleftarrow{\mathrm{id}} S \xrightarrow{\mathrm{id}} S$.

Definition 5 (Typed grammars and typed derivations). A *typed* grammar is a tuple $\mathcal{G} = \langle \mathcal{G}', T, t\rangle$ where $\mathcal{G}' = \langle S, P, \pi\rangle$ is a grammar, $T \in \mathrm{ob}(\mathbf{C})$ is the *type object* and t is the *(rule) typing*, which assigns to each rule name $q \in P \uplus \{\underline{S}\}$ a cocone (span in $\mathbf{C} \downarrow T$) for $\pi(q)$ to T as depicted in the commutative diagram below.

$$\pi(q)\left\{\; L_q \xleftarrow{\alpha_q} K_q \xrightarrow{\beta_q} R_q \right. \qquad t(q)\left\{\; l_q\colon L_q \to T,\ k_q\colon K_q \to T,\ r_q\colon R_q \to T \right.$$

A rule q is called *mono-typed* if l_q and r_q are monos; $\mathcal{G}$ is called *mono-typed* if all $q \in P \uplus \{\underline{S}\}$ are mono-typed.

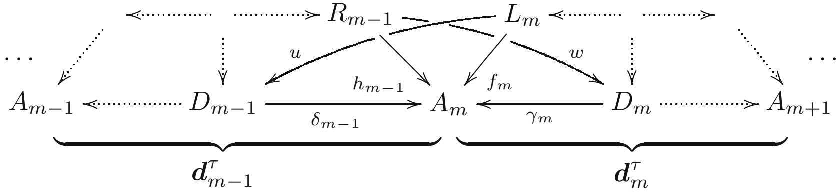
Let $\mathcal{G} = \langle \mathcal{G}', T, t\rangle$ be a typed grammar, where $\mathcal{G}' = \langle S, P, \pi\rangle$; then a *typed $\mathcal{G}$-derivation* is a triple $\rho = \langle \tau, \boldsymbol{d}^\tau, c\rangle$ where $\langle \tau, \boldsymbol{d}^\tau\rangle$ is a $\mathcal{G}'$-derivation and c is a cocone to T for $\boldsymbol{d}^\tau$ that coincides with $t(q)$ on each rule occurrence of q in $\boldsymbol{d}^\tau$ for each $q \in P \uplus \{\underline{S}\}$.



The grammar $\mathcal{G}$ is called *safe* if all objects reachable from the start object are mono-typed.

Consider two rules q_{m-1}, q_m which can be applied in sequence and rewrite A_{m-1} to A_m and then to A_{m+1}, as shown in the next diagram. Furthermore assume that the left-hand side of q_m is already present in D_{m-1} and the right-hand side of q_{m-1} can still be found in D_m. This means that these rules do not interfere with each other and their applications can hence be switched, leading to the same result A_{m+1}. Pairs of direct derivations of this kind are called sequential-independent.

Definition 6 (Sequential independence [7]). Let $\langle \tau, \boldsymbol{d}^\tau \rangle$ be a derivation. Then, fixing $m \in [|\tau|]$, $m > 0$, the direct derivations $\boldsymbol{d}^\tau_{m-1}$ and $\boldsymbol{d}^\tau_m$ are *sequential-independent* if there are morphisms $u \colon L_m \to D_{m-1}$ and $w \colon R_{m-1} \to D_m$ such that the diagram below commutes, i.e., $\delta_{m-1} \circ u = f_m$ and $\gamma_m \circ w = h_{m-1}$.



We shall now introduce certain relations between the rules of a mono-typed grammar, and the resulting connections with sequential independence and the classical Local Church-Rosser Theorem. In the following, the inclusion (or partial order) $\sqsubseteq$, union (or join) $\sqcup$ and intersection (or meet) $\sqcap$ are interpreted in the subobject lattice $\mathrm{Sub}(T)$.

Definition 7 (Rule relations). Let $\mathcal{G} = \langle \langle S, P, \pi \rangle, T, t \rangle$ be a mono-typed grammar and let $q, q' \in P$ be rule names. We define four *rule relations*:

$<$ **:** *q directly causes q'*, written $q < q'$, if $R_q \sqcap L_{q'} \not\sqsubseteq K_q$
$\ll$ **:** *q can be disabled by q'*, written $q \ll q'$, if $L_q \sqcap L_{q'} \not\sqsubseteq K_{q'}$
$<_{\mathsf{co}}$ **:** *q directly co-causes q'*, written $q <_{\mathsf{co}} q'$, if $R_q \sqcap L_{q'} \not\sqsubseteq K_{q'}$
$\ll_{\mathsf{co}}$ **:** *q can be co-disabled by q'*, written $q \ll_{\mathsf{co}} q'$, if $R_q \sqcap R_{q'} \not\sqsubseteq K_q$.

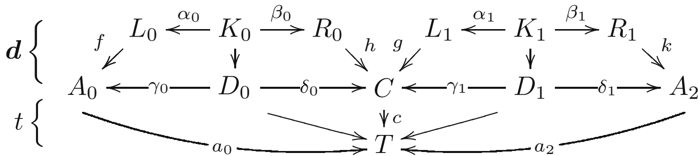
The following proposition gives a partial account of the relationship between sequential independence and rule relations.

Proposition 8. *Let $\langle \tau, \boldsymbol{d}^\tau, c \rangle$ be a typed derivation such that $\boldsymbol{d}^\tau$ witnesses $A_0 \xRightarrow{\langle q_0, f \rangle} C \xRightarrow{\langle q_1, g \rangle} A_2$ and suppose that C is mono-typed. Then:*

1. *If $q_0 \not< q_1$ and $q_0 \not\ll q_1$ then $\boldsymbol{d}^\tau_0$ and $\boldsymbol{d}^\tau_1$ are sequential-independent;*
2. *If $\boldsymbol{d}^\tau_0$ and $\boldsymbol{d}^\tau_1$ are sequential-independent then $q_0 \not< q_1$ and $q_0 \not<_{\mathsf{co}} q_1$.*

As mentioned above, sequential-independent direct derivations can be switched, giving us the first part of the following result. Moreover, when working with mono-typed grammars and derivations, we identify a sufficient condition making it possible to construct the "middle-object" of the switched derivation as a subobject of the type object.

Theorem 9 (Local Church-Rosser). *Consider the typed derivation diagram below:*



where t is a cocone for $\boldsymbol{d}$ to T and assume that the (untyped) direct derivations are sequential-independent. Then the following hold:

1. *There exist C', g', f' and a witness $\boldsymbol{d}'$ for $A_0 \xRightarrow{\langle q_1, g' \rangle} C' \xRightarrow{\langle q_0, f' \rangle} A_2$ such that $\boldsymbol{d}'_0$ and $\boldsymbol{d}'_1$ are sequential-independent.*
2. *If both rules are mono-typed, a_0, c and a_2 are mono, and also $L_0 \sqcap R_1 \sqsubseteq D_0 \sqcap D_1$ in $\mathrm{Sub}(T)$, then $C' = L_0 \sqcup (D_0 \sqcap D_1) \sqcup R_1$.*

Proof. For the the first part of the theorem see [12, 8, 13]. For the second half, let $w_0 \colon R_0 \to D_1$ and $u_0 \colon L_1 \to D_0$ be such that $h = \gamma_1 \circ w_0$ and $g = \delta_0 \circ u_0$.

We obtain the following four diagrams: square (1) by pullback, also yielding pullbacks (2) and (3). Squares (4) and (5) by pushout, also yielding pushouts (6) and (7). Finally, square (8) by pushout. Notice that all the morphisms in the diagrams are mono.

$$\begin{array}{ccccc}
L_0 & \xleftarrow{\alpha_0} & K_0 & \xrightarrow{\beta_0} & R_0 \\
{\scriptstyle u_1}\downarrow & (4) & \downarrow & (2) & \downarrow{\scriptstyle w_0} \\
E_0 & \leftarrow & D_0 \sqcap D_1 & \rightarrow & D_1 \\
{\scriptstyle \gamma'_1}\downarrow & (6) & \downarrow & (1) & \downarrow{\scriptstyle \gamma_1} \\
A_0 & \xleftarrow[\gamma_0]{} & D_0 & \xrightarrow[\delta_0]{} & C
\end{array}
\qquad
\begin{array}{ccccc}
L_1 & \xleftarrow{\alpha_1} & K_1 & \xrightarrow{\beta_1} & R_1 \\
{\scriptstyle u_0}\downarrow & (3) & \downarrow & (5) & \downarrow{\scriptstyle w_1} \\
D_0 & \leftarrow & D_0 \sqcap D_1 & \rightarrow & E_1 \\
{\scriptstyle \delta_0}\downarrow & (1) & \downarrow & (7) & \downarrow{\scriptstyle \delta'_0} \\
C & \xleftarrow[\gamma_1]{} & D_1 & \xrightarrow[\delta_1]{} & A_2
\end{array}$$

$$\begin{array}{ccccc}
L_1 & \xleftarrow{\alpha_1} & K_1 & \xrightarrow{\beta_1} & R_1 \\
{\scriptstyle u_0}\downarrow & (3) & \downarrow & (5) & \downarrow{\scriptstyle w_1} \\
D_0 & \leftarrow & D_0 \sqcap D_1 & \rightarrow & E_1 \\
{\scriptstyle \gamma_0}\downarrow & (6) & \downarrow & (8) & \downarrow{\scriptstyle \gamma'_0} \\
A_0 & \xleftarrow[\gamma'_1]{} & E_0 & \xrightarrow[\delta'_1]{} & C'
\end{array}
\qquad
\begin{array}{ccccc}
L_0 & \xleftarrow{\alpha_0} & K_0 & \xrightarrow{\beta_0} & R_0 \\
{\scriptstyle u_1}\downarrow & (4) & \downarrow & (2) & \downarrow{\scriptstyle w_0} \\
E_0 & \leftarrow & D_0 \sqcap D_1 & \rightarrow & D_1 \\
{\scriptstyle \delta'_1}\downarrow & (8) & \downarrow & (7) & \downarrow{\scriptstyle \delta_1} \\
C' & \xleftarrow[\gamma'_0]{} & E_1 & \xrightarrow[\delta'_0]{} & A_2
\end{array}$$

Notice that $E_0 = L_0 \sqcup (D_0 \sqcap D_1)$ (because a_0 is mono) and $E_1 = R_1 \sqcup (D_0 \sqcap D_1)$ (since a_2 is mono). It remains to show that $C' = E_0 \sqcup E_1$ for which it suffices to show that $E_0 \sqcap E_1 = D_0 \sqcap D_1$. But by assumption $E_0 \sqcap E_1 = (L_0 \sqcap R_1) \sqcup (D_0 \sqcap D_1) = D_0 \sqcap D_1$. □

From a true concurrency point of view, we do not want to distinguish among derivations which differ only in the order of sequential-independent direct derivations. This is formalized by the relation introduced next.

Definition 10 (Derivation switching). Let $\langle \tau, \boldsymbol{d}^\tau \rangle$ be a derivation and assume that the direct derivations $\boldsymbol{d}^\tau_{m-1}$ and $\boldsymbol{d}^\tau_m$ are sequential-independent. Let τ' be the path obtained from τ by switching these two direct derivations according to Theorem 9. Finally let $\boldsymbol{d}^{\tau'}$ be a diagram of τ'. Then we say that the two derivations are *switchings* of each other and write $\langle \tau, \boldsymbol{d}^\tau \rangle \overset{sw}{\sim} \langle \tau', \boldsymbol{d}^{\tau'} \rangle$.

4 Occurrence Grammars

In this section we shall introduce the central notion of *occurrence grammar* which will be used to describe the computation of a system modulo concurrency and on which the notion of process—introduced in Definition 20—relies.

We begin by defining the *asymmetric conflict* relation. It arises in any computational formalism where resources can be read without being consumed. The notion of asymmetric conflict has been previously defined and used for similar purposes in the concrete cases of Petri nets and graph transformation systems. Note that in this paper we deal only with deterministic occurrence grammars.

In the general setting of adhesive grammars, asymmetric conflict can be defined using the rule relations of Definition 7: rules p, q are in asymmetric conflict (written $p \nearrow q$) whenever either p is a (possibly indirect) cause of q or p is disabled by q. In an occurrence grammar every rule occurs exactly once; thus p *must* be executed before q.

Definition 11 (Asymmetric conflict, (co-)causes). Let $\mathcal{G} = \langle\langle S, P, \pi\rangle, T, t\rangle$ be a mono-typed grammar. Then $\nearrow = <^+ \cup (\ll \setminus \mathrm{id}_P)$, where id_P is the identity relation on P, is called *asymmetric conflict*. For a subobject $A \in \mathrm{Sub}(T)$ we define

$$\llcorner A \lrcorner = \{q \in P \mid R_q \sqcap A \not\sqsubseteq K_q\} \quad \text{and} \quad \ulcorner A \urcorner = \{q \in P \mid L_q \sqcap A \not\sqsubseteq K_q\}$$

as the sets of *(direct) causes* and *(direct) co-causes of* A respectively.

We are now ready to define the notion of occurrence grammars. Technically an occurrence grammar is a grammar with special properties which generalizes the notions of deterministic occurrence nets [11] and grammars [4] defined in the setting of Petri nets and graph grammars, respectively.

Definition 12 (Occurrence grammars). A grammar $\mathcal{O} = \langle\langle S, P, \pi\rangle, T, t\rangle$ is a *pre-occurrence grammar* if it is mono-typed,

1. P is finite and the relation $\nearrow$ is acyclic,
2. the start object S has no causes, i.e. $\llcorner S \lrcorner = \varnothing$,
3. there are neither forward nor backward conflicts, i.e., for all $q \neq q' \in P$

$$(L_{q'} \sqcap L_q) \sqsubseteq K_{q'} \sqcup K_q \quad \text{and} \quad (R_{q'} \sqcap R_q) \sqsubseteq K_{q'} \sqcup K_q.$$

A pre-occurrence grammar $\mathcal{O}$ is said to be an *occurrence grammar* if also:

4. there is an *end object* $F \in \mathrm{Sub}(T)$ such that $\ulcorner F \urcorner = \varnothing$;
5. for all subobjects $A \in \mathrm{Sub}(T)$

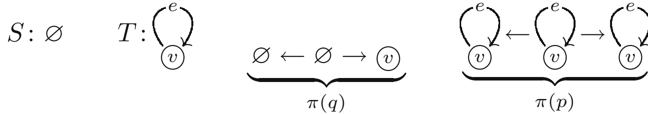
$$(a) \quad A \sqsubseteq \left(S \sqcup \bigsqcup_{q \in \llcorner A \lrcorner} R_q\right) \quad \text{and} \quad (b) \quad A \sqsubseteq \left(F \sqcup \bigsqcup_{q \in \ulcorner A \urcorner} L_q\right).$$

The requirements of Definition 12 above can be motivated as follows: First, $\nearrow$ must be acyclic, since there is otherwise no valid execution order for all rules of the occurrence grammar. Furthermore there are no forward conflicts, meaning that the occurrence grammar is deterministic, and no backward conflicts which roughly amounts to saying that "everything" is generated by at most one rule. It can be shown that S and F are uniquely determined by the axiom 5 of Definition 12.

Indeed, the axiom 5 is central for the following theory. It intuitively says that "everything" is either in the start object or generated at some point and that also the converse holds: "everything" is either in the end object or it is consumed at some time. The first part is needed to show that when we put the rules of an occurrence grammar into sequence according to asymmetric conflict and apply

an initial part of this sequence, we reach an object that contains the left-hand side of the next rule. Then the second part is needed to prove that also the pushout complement exists and thus the rule can actually be applied. (See also the proof of Theorem 19.) Its role is further explained by the example below.

Example 13 (Pre-occurrence grammar that is not an occurrence grammar). Consider the adhesive category of graphs and graph homomorphisms, **Graph**. Now take a grammar with the empty graph $\varnothing$ as start object S, and two rule names p, q with associated rules and type graph as shown below.



The typing is given by the obvious inclusions. This is clearly a pre-occurrence grammar, but not an occurrence grammar since axiom 5(a) of Definition 12 is violated. Indeed, $\llcorner T \lrcorner = \{q\}$ and thus $T \not\sqsubseteq S \sqcup \bigsqcup_{q' \in \llcorner T \lrcorner} R_{q'} = S \sqcup R_q = R_q$. Note that this corresponds to the fact that the graph obtained after applying q is *too small* to contain the left-hand side of p.

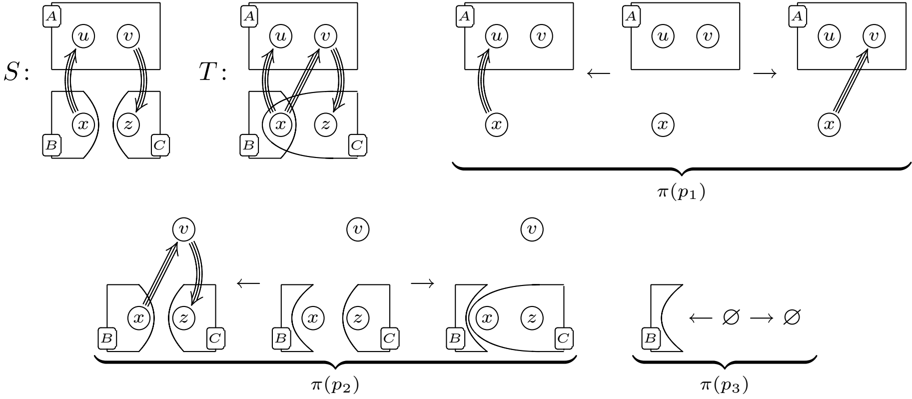
Similarly, when we consider the reversed pre-occurrence grammar (view rules from right to left) with T as the start object, axiom 5(b) of Definition 12 does not hold. In order to see this observe that now the end object is the empty graph and that only (the reversed) q is a co-cause for T, which leads to $T \not\sqsubseteq F \sqcup \bigsqcup_{q' \in \ulcorner A \urcorner} L_{q'}$. This is related to the fact that—after applying rule p (reversely) to T—q cannot be applied since the pushout complement for $\varnothing \rightarrowtail \circ \rightarrowtail$ ⟲∘ does not exist, due to the presence of the edge.

In previous approaches, the subobject inclusions followed indirectly from axioms about individual items. For instance [5] defines a deterministic occurrence grammar $\mathcal{O}$ requiring that whenever a node v is deleted by a rule of $\mathcal{O}$ and an edge e attached to v is created by $\mathcal{O}$, then $\mathcal{O}$ must also delete e.

Example 14 (Graphs with scopes). In order to show that our theory applies to a setting wider than standard graph rewriting, we consider graphs with scopes where each node is contained in a set of scopes. These graphs can be viewed as objects of the functor category $\mathbf{Set_{fin}}^{\bullet \leftarrow \bullet \rightarrow \bullet \leftleftarrows \bullet}$, which is adhesive. Concretely, every object consists of a set of nodes V, a set of edges E, a set of scopes S and an auxiliary set X, used to relate nodes and scopes. We have functions $src, tgt \colon E \to V$, $sc_S \colon X \to S$, $sc_V \colon X \to V$. If there is an element $x \in X$ with $sc_S(x) = s \in S$ and $sc_V(x) = v \in V$ we say that v is contained in or within scope s. A node may belong to several scopes and a scope may contain several nodes. We draw the graph part of the objects in the usual way. Scopes are depicted by labelled boxes around the nodes they contain (see below).

The following example grammar is inspired by scope extrusion in process calculi. We want to model that a node is moved from one scope into another by a reaction rule. The first rule (p_1) can move the target of an edge within the same scope, the second (p_2) is a reaction where a node v is transferred from one

scope to another whenever there is a two-edge path from it to a node w within the second scope, and the third (p_3) models garbage collection of empty scopes. Note that rule (p_3) cannot be applied to non-empty scopes, since the pushout complement of diagram (1) in Definition 4 would not exist, intuitively because the removal of the scope would leave some dangling links.



By taking S and T above as the start and type graph respectively, and the obvious inclusions as rule typings we obtain an occurrence grammar where p_1 is a cause for p_2 ($p_1 < p_2$) and p_2 is in asymmetric conflict with p_3 ($p_2 \nearrow p_3$).

After these motivating examples, we will continue to develop the theory. First we show that if every rule is applied at most once then the reached object is mono-typed. A consequence of this is that any object reachable in a consuming pre-occurrence grammar is mono-typed.

Proposition 15 (Quasi-safety and safety of consuming grammars). *Let $\mathcal{O} = \langle\langle S, P, \pi\rangle, T, t\rangle$ be a pre-occurrence grammar. Then for each path $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ and typed derivation $\rho = \langle \tau, \boldsymbol{d}^\tau, c\rangle$, with $\boldsymbol{d}^\tau$ witnessing $S \stackrel{\tau}{\Rightarrow} A_n$, if no rule occurs twice in τ then*

1. *A_n is mono-typed, i.e., c_{A_n} is a mono,*
2. *asymmetric conflict is respected, i.e. $\forall i, j \in [n] \,.\, q_i \nearrow q_j \Rightarrow i < j$,*
3. *the inclusion cocone to $S \sqcup \bigsqcup_{i\in[m]} R_i$ for $m \leq n$ is a colimit of $\boldsymbol{d}^\tau_{[m]}$.*

In particular, if $\mathcal{O}$ is consuming then any rule can be applied at most once in each typed $\mathcal{O}$-derivation and thus 1–3 above hold for any typed derivation.

Another fact that holds in the setting of pre-occurrence grammars is that all typed derivations which apply the same rules, possibly in different order, are equivalent when seen as truly concurrent computations. Formally, this involves the notion of switch-equivalence for typed derivations.

Definition 16 (Switch equivalence). Let $\rho = \langle \tau, \boldsymbol{d}^\tau, c\rangle$ and $\rho' = \langle \tau', \boldsymbol{d}^{\tau'}, c'\rangle$ be two typed $\mathcal{G}$-derivations, with $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ and $\tau' = \langle q'_i, f'_i\rangle_{i\in[n]}$. Then ρ and ρ' are *isomorphic*, written $\rho \cong \rho'$, if $q_i = q'_i$ for each $i \in [n]$ and there

is a diagram isomorphism $\iota\colon \langle \boldsymbol{d}^{\tau}, c\rangle \cong \langle \boldsymbol{d}^{\tau'}, c'\rangle$ that relates the start object, rule-occurrences and the type objects by identities.

Moreover $\rho \overset{sw}{\sim} \rho'$ if $\langle \tau, \boldsymbol{d}^{\tau}\rangle \overset{sw}{\sim} \langle \tau', \boldsymbol{d}^{\tau'}\rangle$ and finally *switch-equivalence* $\overset{sw}{\approx}$ is the union of the transitive closure of $\overset{sw}{\sim}$ and $\cong$, in signs $\overset{sw}{\approx} = (\overset{sw}{\sim})^* \cup \cong$.

Lemma 17 (Switch equivalence in pre-occurrence grammars). *Let* $\mathcal{O} = \big\langle \langle S, P, \pi\rangle, T, t\big\rangle$ *be a pre-occurrence grammar, and let* $\rho = \langle \tau, \boldsymbol{d}^{\tau}, c\rangle$ *and* $\rho' = \langle \tau', \boldsymbol{d}^{\tau'}, c'\rangle$ *be typed* $\mathcal{O}$*-derivations where* $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ *and* $\tau' = \langle q_i', f_i'\rangle_{i\in[n]}$ *are paths in which no rule occurs twice and* $\big\langle q_i\big\rangle_{i\in[n]}$ *is a permutation of* $\big\langle q_i'\big\rangle_{i\in[n]}$*. Then the two typed derivations are switch-equivalent, i.e.,* $\rho \overset{sw}{\approx} \rho'$*.*

The above facts about pre-occurrence grammars have a premise about the existence of some derivation. In the context of proper occurrence grammars we can single out sufficient conditions for the existence of derivations, which can be described in terms of asymmetric conflict $\nearrow$.

Definition 18 (Rule linearizations). Let $\mathcal{O} = \big\langle \langle S, P, \pi\rangle, T, t\big\rangle$ be a pre-process and let $P' \subseteq P$ and $n = |P'|$. Then a sequence $\boldsymbol{q} = \big\langle q_i\big\rangle_{i\in[n]} \in (P')^*$ is a *(rule) linearization of* P' if $P' = \{q_i \mid i \in [n]\}$ and $\forall i, j \in [n] \,.\, q_i \nearrow q_j \Rightarrow i < j$. The set of all linearizations of P' is denoted by $\mathrm{lin}(P')$ and $\boldsymbol{q}_i = q_i$ by convention.

We write $S \overset{\boldsymbol{q}}{\Rightarrow} A$ if $S \overset{\tau}{\Rightarrow} A$ and $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ is a path for some sequence of matches $\big\langle f_i\big\rangle_{i\in[n]}$.

The next theorem gives two central results. Firstly, if $\mathcal{O}$ is an occurrence grammar, then there exists a typed derivation which rewrites the start object into the end object, applying all the rules in any order that respects asymmetric conflict. Secondly, if the type object is not too large and there exists a linearization of all rules that leads to a typed derivation then a pre-occurrence grammar is an occurrence grammar.

Theorem 19. *Let* $\mathcal{O}$ *be a pre-occurrence grammar.*

1. *If* $\mathcal{O}$ *is an occurrence grammar, then* $\forall \boldsymbol{q} \in \mathrm{lin}(P).\, S \overset{\boldsymbol{q}}{\Rightarrow} F$*, where* F *is the end object of* $\mathcal{O}$*.*
2. *If* $\exists \boldsymbol{q} \in \mathrm{lin}(P).\, \exists F \in \mathrm{Sub}(T).\, S \overset{\boldsymbol{q}}{\Rightarrow} F$ *and* $T = S \sqcup \bigsqcup_{q\in P} R_q$*, then* $\mathcal{O}$ *is an occurrence grammar.*

Proof (idea). The crucial point is the proof of the first part, i.e., of the fact that any linearization of P gives rise to a typed derivation. Let $\boldsymbol{q} = \boldsymbol{p}q\boldsymbol{p}' \in \mathrm{lin}(P)$ and assume that $S \overset{\boldsymbol{p}}{\Rightarrow} A$. Then we have to show that $L_q \sqsubseteq A$ and that the pushout complement for $A \leftarrowtail L_q \overset{\alpha}{\leftarrowtail} K_q$ exists.

By using axiom 5(a) of Definition 12 we can prove that A is the greatest object with causes in $\boldsymbol{p}$ and co-causes in $\boldsymbol{p}'$. Then $L_q \sqsubseteq A$ follows immediately. It remains to show that the pushout complement exists: the candidate is $\tilde{D} = (S \sqcup \bigsqcup_{q\in \boldsymbol{p}} R_q) \sqcap (F \sqcup \bigsqcup_{q\in \boldsymbol{p}'} L_q)$.

By using only facts about pre-occurrence grammars one can show that $\tilde{D}$ is the greatest subobject of A which forms a pullback together with the arrows $A \leftarrowtail L_q \overset{\alpha}{\leftarrowtail} K_q$. Finally, using axiom 5(b) of Definition 12 and some elementary category theory we can show that $\tilde{D}$ is actually a pushout complement. □

An interesting point of the proof is that the question about the existence of pushout complements can be answered in lattice-theoretic terms only.

5 From Derivations to Processes and Back

We now come to the one-to-one correspondence between switch-equivalence classes of derivations and processes. After introducing the notion of process (for a given grammar), we show that such a process can be seen as a representative of a full class of switch-equivalent typed derivations, all of which are linearizations of the process. Vice versa, given a derivation, a colimit-based construction allows to derive a corresponding process. The result states that these two constructions are (essentially) inverse to each other.

We shall now define the notion of process, i.e., a truly concurrent computation of a specific grammar $\mathcal{G}$ represented by an occurrence grammar.

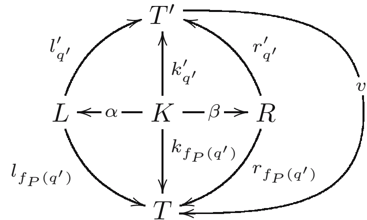
Definition 20 (Processes). Let $\mathcal{G} = \big\langle \langle S, P, \pi \rangle, T, t \big\rangle$ be a grammar. Then a $\mathcal{G}$*-process* is a triple $\mathcal{P} = \langle \mathcal{O}, v, f_P \rangle$ where $\mathcal{O} = \big\langle \langle S', P', \pi' \rangle, T', t' \big\rangle$ is an occurrence grammar and

- $v \colon T' \to T$ is a morphism between the type objects, and
- $f_P \colon P' \uplus \{\underline{S'}\} \to P \uplus \{\underline{S}\}$ is a function between rule names with $f_P(\underline{S'}) = \underline{S}$

such that for all $q' \in P' \uplus \{\underline{S'}\}$

1. $\pi'(q') = \pi(f_P(q'))$
2. and[2] $v \circledcirc t'(q') = t(f_P(q'))$

i.e. the diagram on the right commutes, where $\pi'(q') = L \overset{\alpha}{\leftarrowtail} K \overset{\beta}{\rightarrowtail} R = \pi(f_P(q'))$.



Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two $\mathcal{G}$-processes. An *isomorphism* $\langle i, j \rangle \colon \mathcal{P}_1 \cong \mathcal{P}_2$ from $\mathcal{P}_1$ to $\mathcal{P}_2$ is a pair $\langle i, j \rangle$ such that (i) $\big\langle \mathcal{O}_1, i, j \big\rangle$ is an $\mathcal{O}_2$-process, (ii) $i \colon T_1 \to T_2$ is an isomorphism satisfying $v_2 \circ i = v_1$, and (iii) $j \colon P_1 \uplus \{\underline{S_1}\} \to P_2 \uplus \{\underline{S_2}\}$ is a bijection satisfying $f_{P1} = f_{P2} \circ j$.

Intuitively, an occurrence grammar $\mathcal{O}$ only represents an "autonomous" concurrent computation, whereas the pair $\langle v, f_P \rangle$ provides a link back to a grammar. The morphism v specifies how such a computation can be "typed" over the type object of $\mathcal{G}$, and f_P specifies how the rule occurrences of $\mathcal{O}$ can be seen as instances of rules in $\mathcal{G}$.

[2] For a cocone c to an object A and a morphism $v \colon A \to B$ we denote by $v \circledcirc c$ the cocone to B obtained by composing every morphism in c with v.

Given a process $\mathcal{P}$ of a grammar $\mathcal{G}$, we can obtain a corresponding typed derivation in $\mathcal{G}$ by taking any linearization of the rules in $\mathcal{O}$, applying each such rule in the specified order (possible by Theorem 19) and retyping the generated derivation over the type object of $\mathcal{G}$.

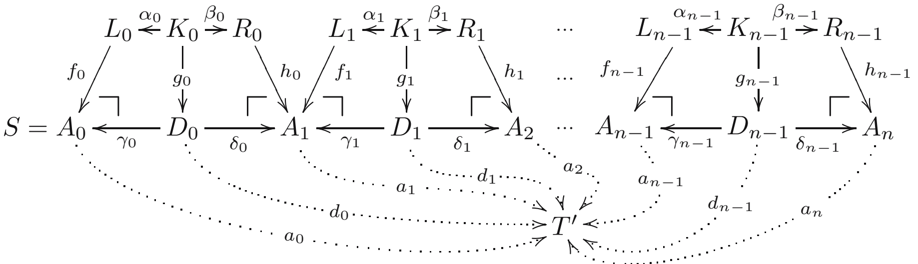
Definition 21 (Drv—**derivations of a process).** Let $\mathcal{P} = \langle \mathcal{O}, v, f_P \rangle$ be a $\mathcal{G}$-process, where $\mathcal{O} = \langle \langle S, P, \pi \rangle, T', t' \rangle$. Let $\boldsymbol{q} \in \text{lin}(P)$ be a linearization of P and let $\rho = \langle \tau, \boldsymbol{d}^\tau, c \rangle$ be a typed derivation witnessing $S \overset{\boldsymbol{q}}{\Rightarrow}_{\mathcal{O}} F$. Then $\langle \tau, \boldsymbol{d}^\tau, v \circledcirc c \rangle$ is called a typed $\mathcal{P}$-*derivation.* The set of all such derivations is denoted by $\text{Drv}(\mathcal{P})$.

The next proposition shows that all derivations of a given process are "equivalent" from a true concurrency point of view. Hence Drv induces a mapping from (isomorphism classes of) processes to switch-equivalence classes of derivations.

Proposition 22. *Let $\mathcal{P}$ and $\mathcal{P}'$ be processes such that $\mathcal{P} \cong \mathcal{P}'$. Then for all $\rho \in \text{Drv}(\mathcal{P})$ and $\rho' \in \text{Drv}(\mathcal{P}')$ it holds $\rho \overset{sw}{\approx} \rho'$.*

Vice versa, given any derivation in a grammar $\mathcal{G}$, we can generate a corresponding process as follows. The colimit of the (untyped part) of the derivation diagram is the type object of the process, while the rule instances of the derivation become the rules of the process. The morphism back to the type object of $\mathcal{G}$ is given by the mediating morphism to the typed derivation cocone. The next definition describes this procedure formally.

Definition 23 (Prc—**processes of a derivation).** Let $\tau = \langle q_i, f_i \rangle_{i \in [n]}$ be a path and $\rho = \langle \tau, \boldsymbol{d}^\tau, c \rangle$ be a typed derivation for a grammar $\mathcal{G} = \langle \langle S, P, \pi \rangle, T, t \rangle$. Let $\bar{c}$ be a colimit cocone for $\boldsymbol{d}^\tau$ to T', whose components are the dotted arrows below.



Define $\mathcal{O} = \langle \langle S', P', \pi' \rangle, T', t' \rangle$ to be a grammar where:

- $S' = S$;
- $P' = \{ \langle q_i, i \rangle \mid i \in [n] \wedge \tau_i = \langle q_i, f_i \rangle \}$ is a set that contains a *rule occurrence name* for each rule occurrence of $\boldsymbol{d}^\tau$, and
- π' with $\pi'(\langle q_i, i \rangle) = \pi(q_i)$ assigns each rule occurrence name the rule of the grammar $\mathcal{G}$ it originates from; and
- $t'(\langle q_i, i \rangle)$ is a cocone for $\pi(q_i)$ to T', which gives the typing for each rule occurrence $\langle q_i, i \rangle \in P'$ as indicated below

$$\begin{array}{l} \pi'(\langle q_i, i\rangle)\{ \\ t'(\langle q_i, i\rangle)\{ \end{array} \quad L_i \xleftarrow{\alpha_i} K_i \xrightarrow{\beta_i} R_i, \quad L_i \xrightarrow{a_i \circ f_i} T', \quad K_i \xrightarrow{d_i \circ g_i} T', \quad R_i \xrightarrow{a_{i+1} \circ h_i} T'$$

and $t'(\underline{S'})$ is the cocone obtained by taking three times morphism a_0.

Finally let $v\colon T' \to T$ be the mediating morphism from the colimit $\bar{c}$ to the cocone c. Then

$$\mathcal{P} = \langle \mathcal{O}, v, f_P\colon P' \uplus \{\underline{S'}\} \to P \uplus \{\underline{S}\} \rangle$$

with $f_P(\langle q_i, i\rangle) = q_i$ and $f_P(\underline{S'}) = \underline{S}$ is a *ρ-process*. The set of all ρ-processes—all of them being isomorphic to each other—is denoted by $\mathrm{Prc}(\rho)$.

The next proposition shows that starting from switch-equivalent derivations, the construction described in Definition 23 produces isomorphic processes. Hence Prc can be seen as a function from switch-equivalence classes of derivations to isomorphism classes of processes.

Proposition 24. *Let ρ and ρ' be typed $\mathcal{G}$-derivations such that $\rho \overset{sw}{\approx} \rho'$. Then $\mathcal{P} \cong \mathcal{P}'$ holds for all $\mathcal{P} \in \mathrm{Prc}(\rho)$ and $\mathcal{P}' \in \mathrm{Prc}(\rho')$.*

We conclude with the main result of this section, stating that Prc and Drv can be seen as functions between switch-equivalence classes of derivations and isomorphism classes of processes, and that they are inverse to each other.

Theorem 25. *Let ρ be a typed $\mathcal{G}$-derivation and $\mathcal{P}$ be a $\mathcal{G}$-process. Then:*

$$\begin{array}{llll} 1. & \rho' \in \mathrm{Drv}\big(\mathrm{Prc}(\rho)\big) & \textit{implies} & \rho' \overset{sw}{\approx} \rho \\ 2. & \mathcal{P}' \in \mathrm{Prc}\big(\mathrm{Drv}(\mathcal{P})\big) & \textit{implies} & \mathcal{P}' \cong \mathcal{P} \end{array}$$

6 Conclusion

We have shown that the notion of process, originally introduced for Petri nets, can be studied in the general setting of DPO rewriting systems over adhesive categories. This is theoretically pleasing, since it allows one to study this fundamental concept at the same abstract level as, for instance, the notion of sequential-independence.

While the fact that processes can be studied in an abstract framework may not seem surprising, the generalization is non-trivial to obtain. The reason is that the previous definitions of occurrence grammars and processes, e.g. of Petri nets and graph grammars, used the inherently set-theoretical concept of items: atomic units that are consumed and produced. The absence of an analogous concept for adhesive categories has required the development of original techniques, mainly relying on the algebra of the subobject lattice of the type object.

As a consequence of its generality, the theory developed in this paper is applicable to a wide range of rewriting systems. It enables us to handle various graph-like structures which appear in the literature and are used in tools.

While starting the development of an encompassing theory of true concurrency, we have also laid the foundations for the use of partial order verification techniques. Specifically, the generalization of methods developed for Petri nets and graph transformation systems (see, e.g., [14, 10, 2, 3]) appears as a stimulating direction of research. In order to achieve this goal, future work will concern *unfoldings*: non-deterministic (infinite) processes which fully describe the behavior of a system.

References

1. P. Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 2000.
2. P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, volume 2154 of *LNCS*, pages 381–395. Springer Verlag, 2001.
3. P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR 2004*, volume 3170 of *LNCS*, pages 83–98. Springer Verlag, 2004.
4. P. Baldan, A. Corradini, and U. Montanari. Concatenable graph processes: relating processes and derivation traces. In *Proc. ICALP'98*, volume 1443 of *LNCS*. Springer Verlag, 1998.
5. P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In *Proc. of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer Verlag, 1999.
6. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.
7. H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Proceedings of the 1st International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer Verlag, 1979.
8. H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Mathematical Structures in Computer Science*, 1:361–404, 1991.
9. H. Ehrig, A. Habel, J. Padberg, and U. Prange. Adhesive high-level replacement categories and systems. In *Proc. of ICGT'04*, volume 3256 of *LNCS*, pages 144–160. Springer Verlag, 2004.
10. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(20):285–310, 2002.
11. U. Goltz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983.
12. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
13. S. Lack and P. Sobociński. Adhesive and quasiadhesive categories. *Theoretical Informatics and Applications*, 39(2):511–546, 2005.
14. K.L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

On Metric Temporal Logic and Faulty Turing Machines

Joël Ouaknine and James Worrell

Oxford University Computing Laboratory, UK
{joel, jbw}@comlab.ox.ac.uk

Abstract. Metric Temporal Logic (MTL) is a real-time extension of Linear Temporal Logic that was proposed fifteen years ago and has since been extensively studied. Since the early 1990s, it has been widely believed that some very small fragments of MTL are undecidable (i.e., have undecidable satisfiability and model-checking problems). We recently showed that, on the contrary, some substantial and important fragments of MTL are decidable [19, 20]. However, until now the question of the decidability of full MTL over infinite timed words remained open.

In this paper, we settle the question negatively. The proof of undecidability relies on a surprisingly strong connection between MTL and a particular class of faulty Turing machines, namely 'insertion channel machines with emptiness-testing'.

1 Introduction

The theory of automated verification in the untimed world has by now achieved a respectable maturity: there is a plethora of modelling and specification formalisms, with well-understood associated algorithms—see, e.g., [22] for a comprehensive survey of the field. Over the past two decades, many researchers have attempted to extend this methodology to the real-time world, in which *quantitative* timing constraints are of interest. One of the earliest and most prominent real-time specification formalisms to be proposed was Metric Temporal Logic (MTL) [16, 6], which extends Linear Temporal Logic (LTL) in that the various temporal operators are annotated with time intervals. For example, whereas the LTL formula $\Box(req \implies \Diamond grant)$ specifies that every *req* is always eventually followed by a *grant*, the MTL formula $\Box(req \implies \Diamond_{[3,5]} grant)$[1] specifies in addition that the *grant*s shall happen *within* 3 *to* 5 *time units of the occurrence of each req*. This type of *bounded-response* property arises naturally when considering safety-critical systems such as a car's braking system, or a power plant's shutdown mechanism.

MTL formulas are usually interpreted over *dense time*, which is typically modelled using the non-negative real numbers $\mathbb{R}_{\geq 0}$.[2] Furthermore, an important

[1] The $\Box$ operator is here implicitly annotated with the interval $[0, \infty)$.

[2] By contrast, in *discrete-time* settings the underlying model is typically the non-negative integers, yielding more tractable theories that however correspond less closely to physical reality [13, 3].

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 217–230, 2006.
© Springer-Verlag Berlin Heidelberg 2006

distinction among real-time models is whether one assumes that the system under consideration is observed at every instant in time, leading to an *interval-based semantics* [4, 21, 14], or whether one only records a countable sequence of snapshots of the system, leading to a *point-based semantics* [11, 6, 7, 12, 13, 24]. The interval-based semantics is somewhat more intuitive if one interprets atomic MTL formulas as *state propositions*, whereas the point-based semantics lends itself more naturally to the interpretation of atomic MTL formulas as instantaneous *events* or *actions*. Our main (undecidability) result concerns the point-based semantics, and accordingly that is the semantics we focus on in this paper. As it turns out, the corresponding undecidability result in the interval-based setting has been known for quite some time; see, e.g., [11, 5].

As is the case for LTL, it is possible to extend MTL with *past* temporal operators, although this variant is seldom seen in the literature. MTL with past operators, in turn, is subsumed by a certain monadic logic of timed state sequences introduced in [6]. The satisfiability problem for this logic is shown in that paper to be undecidable. The idea is to encode the halting computations of a given Turing machine as a set of timed words, which can themselves be captured by a monadic formula: configurations of the machine can be encoded within a single unit-duration time interval, since the density of time can accommodate arbitrarily large tape contents. The formula need only specify that the configurations are accurately propagated from one time interval to the next. As a result, the formula is satisfiable iff the Turing machine has a halting computation.

This construction easily carries over to MTL *with past operators*, and in fact the key ingredient required is merely *punctuality*: the ability to specify that two events occur exactly one time unit apart from each other. Unfortunately, a small oversight led to the claim, subsequently reproduced many times—see, e.g., [5, 6, 12, 15], among others—that any logic strong enough to express forward punctuality, i.e., formulas of the form $\Box(p \implies \Diamond_{[1,1]} q)$, is automatically undecidable.

In [19] we showed this claim to be erroneous by proving that the satisfiability problem for MTL over finite timed words is decidable. Recently, we also showed that the *safety* fragment of MTL, in which all 'eventuality' operators are time-bounded, is also decidable [20]. Another important decidability result appears in [4, 21, 14], where the fragment of MTL that disallows singular intervals is proved to be decidable. Yet another decidability result for a fragment of MTL can be found in [13], exploiting digitization techniques.

In light of these developments, the question of the decidability of (standard) MTL, incorrectly considered settled for many years, took on a new urgency. This paper closes the gap by showing that the (infinite) satisfiability and model-checking problems for MTL are undecidable. The proof proceeds by establishing a strong connection between MTL formulas and a particular class of faulty Turing machines, namely *insertion channel machines with emptiness-testing*, or ICMET. Using this connection, satisfiability questions about MTL formulas can be translated back and forth to 'recurrent-state problems' for ICMETs. We show the latter to be undecidable in general from first principles.

Our undecidability result also ties up a couple of loose ends. In [19], for instance, we show that MTL formulas can be encoded into one-clock timed alternating automata (see also [18]) with a weak parity acceptance condition. MTL satisfiability then corresponds to the non-emptiness problem for these automata, which this paper therefore shows to be undecidable. Our present results also imply the undecidability of universality for one-clock Büchi timed automata [2, 17], since these can easily capture (timed encodings of) the non-recurrent/invalid computations of ICMETs, following an idea similar to that used in [3].

2 Faulty Turing Machines

A *channel machine* [1, 9, 23] consists of a finite-state automaton acting on a finite number of unbounded fifo channels (queues, buffers). We are interested in a particular type of channel machines, which we call *insertion channel machines with emptiness-testing*, or *ICMET*. An ICMET is a tuple $\mathcal{C} = (S, \mathit{init}, M, C, \Delta)$, where S is a finite set of *control states*, $\mathit{init} \in S$ is the *initial control state*, M is a finite set of *messages*, C is a finite set of *channels*, and $\Delta \subseteq S \times L \times S$ is the transition relation over label set $L = \{c!m, c?m, c{=}\emptyset \mid c \in C \wedge m \in M\}$.

Intuitively, a $c!m$-transition corresponds to writing m to the tail of channel c, a $c?m$-transition corresponds to reading m from the head of channel c, whereas a $c{=}\emptyset$-transition is only enabled if channel c is empty. The latter transitions, which we call *emptiness-testing*, are useful in the presence of *insertion errors*, as we explain shortly.

A *global state* of an ICMET $\mathcal{C}$ is a tuple $(s, \overline{x})$, where $s \in S$ is the control state and $\overline{x} \in (M^*)^C$ represents the contents of all the channels. We write x_c to denote the contents of a given channel c. The rules in Δ induce an L-labelled transition relation on the set of global states as follows: $(s, c!m, t) \in \Delta$ yields a transition $(s, \ldots, x_c, \ldots) \xrightarrow{c!m} (t, \ldots, x_c{\cdot}m, \ldots)$ that writes $m \in M$ to the tail of channel c, and leaves all other channels unchanged. Likewise, $(s, c?m, t) \in \Delta$ yields a transition $(s, \ldots, m{\cdot}x_c, \ldots) \xrightarrow{c?m} (t, \ldots, x_c, \ldots)$ that reads $m \in M$ from the head of channel c, and again leaves all other channels unchanged. Finally, $(s, c{=}\emptyset, t) \in \Delta$ yields a transition $(s, \overline{x}) \xrightarrow{c=\emptyset} (t, \overline{x})$, provided that channel c is empty, i.e., $x_c = \varepsilon$.

If the above transitions were the only ones allowed, then $\mathcal{C}$ would be an *error-free* channel machine. An ICMET, however, may suffer from insertion errors, which are represented by certain additional transitions.

Given $x, y \in M^*$, write $x \sqsubseteq y$ if x can be obtained from y by deleting any number of letters. For example, HIGMAN $\sqsubseteq$ HIGHMOUNTAIN, as indicated by the underlining. Extend this relation to $(M^*)^C$ by writing $\overline{x} \sqsubseteq \overline{y}$ if, for all $c \in C$, $x_c \sqsubseteq y_c$.

Insertion errors are then introduced by extending the transition relation on global states with the following clause: if $(s, \overline{x}) \xrightarrow{\alpha} (t, \overline{y})$, $\overline{x}' \sqsubseteq \overline{x}$, and $\overline{y} \sqsubseteq \overline{y}'$, then $(s, \overline{x}') \xrightarrow{\alpha} (t, \overline{y}')$.

A *computation* of $\mathcal{C}$ is a finite or infinite sequence of transitions between global states $(s_0, \overline{x_0}) \xrightarrow{\alpha_0} (s_1, \overline{x_1}) \xrightarrow{\alpha_1} (s_2, \overline{x_2}) \xrightarrow{\alpha_2} \cdots$, with $s_0 = init$.[3]

Note 1. Channel machines with insertion errors were first considered in [9], and later used to obtain complexity lower bounds for real-time verification problems in [19, 2]. Emptiness-testing is a well-known computational device (used, for example, in counter machines), which however adds no intrinsic power to *error-free* channel machines. In the presence of insertion errors, emptiness-testing provides a restricted amount of error detection, yet this combination has, to the best of our knowledge, never been studied before. As Theorem 2 indicates, the resulting class ICMET has computational power strictly between that of channel machines with insertion errors and that of perfect channel machines, and turns out to be a useful tool to study Metric Temporal Logic.

We are interested in the following decision problems concerning ICMETs. Let $\mathcal{C} = (S, init, M, C, \Delta)$ be an arbitrary ICMET, and let $t \in S$ be a particular control state of $\mathcal{C}$. The *halting problem* (also known as the *control-state reachability problem*) asks whether there is a computation of $\mathcal{C}$ that reaches t (irrespective of the contents of the channels). The *recurrent-state problem*, on the other hand, asks whether $\mathcal{C}$ has an infinite computation that visits t infinitely often (again, irrespective of channel contents).

Theorem 2. *The halting problem for ICMETs is decidable, with non-primitive recursive complexity. The recurrent-state problem for ICMETs is undecidable.*

Note that both problems are undecidable for error-free channel machines, since these are Turing-powerful. On the other hand, both problems are trivially decidable (with polynomial-time complexity) for channel machines with insertion errors (but without emptiness-testing), since insertion errors make the contents of channels irrelevant (all read- and write-transitions of every control state are at all times enabled)—see [9]. We conclude that *emptiness-testing* imparts a genuine amount of computational power to channel machines with insertion errors, which however falls short of that of perfect channel machines.

Proof. The proof of decidability relies on the theory of well-structured transition systems [10], whereas the complexity lower bound is a corollary of Proposition 25 of [19], which itself makes use of a result of Schnoebelen [23]. Both proofs are omitted here for reasons of space.

For the purposes of this paper the most important result is the undecidability of the recurrent-state problem, and accordingly we now present the proof in detail.

Let $\mathcal{C} = (S, init, M, C, \Delta)$ be an ICMET. Let $m \in M$ be a message and $c \in C$ be a channel. We would first like to define a 'macro' operation, called *occurrence-testing*, that succeeds only if c does not comprise any occurrence of m.

---

[3] One might in addition require that $\overline{x_0} = (\varepsilon, \ldots, \varepsilon)$, but in the presence of insertion errors this constraint is pointless.

To this end, assume that $\mathcal{C}$ has an extra working channel, called *temp*. To perform occurrence-testing for message m on channel c, do the following:

1. Repeatedly read off messages from c and copy them onto *temp*; if any of these messages turn out to be m, halt.
2. At some point, nondeterministically do an emptiness test on c, i.e., proceed if c is empty, otherwise halt. This guarantees that the whole of c has been copied onto *temp*.
3. Copy back the contents of *temp* onto c, ascertaining success by doing an emptiness test on *temp*.

Bearing in mind that insertion errors can occur at any time, the only conclusions that can be drawn from a successful '$m \notin c$'-occurrence-test are that (i) immediately prior to performing occurrence-testing, m did not occur within c, and (ii) upon completing occurrence-testing, c comprises at least all of its original contents, in the right order.

In what follows, occurrence-testing will repeatedly be invoked as if it were a *bona fide* atomic operation. In fact, we will also perform occurrence-testing for sets of messages, to be understood as a sequence of occurrence-tests for each element.

Let $\mathcal{T}$ be a deterministic one-tape Turing machine with tape alphabet Σ. Assume that in any infinite computation of $\mathcal{T}$ the tape contents grows unboundedly. (If this is not outright the case, simply augment $\mathcal{T}$ with a counter that periodically gets incremented.) For technical reasons, assume also that once the tape head visits a particular cell, that cell is never blank afterwards (a blank cell is represented by the symbol $B \in \Sigma$; the assumption is therefore that B can only be read by the head, but not written). The (suitably defined) halting problem for $\mathcal{T}$, when starting on a blank tape, is well-known to be undecidable.

It is equally well-known that a Turing machine such as $\mathcal{T}$ can easily be simulated by an *error-free* channel machine [8]. A single channel is required, which is used to mimic the tape of $\mathcal{T}$. The set M of channel messages includes $\{a, \widehat{a} \mid a \in \Sigma\}$. The 'hatted' versions of the symbols are used to indicate the current position of the head on the tape—accordingly, a channel should always comprise exactly one hatted symbol, except perhaps during the simulation of a head transition. M may contain other messages, to keep track, for example, of the leftmost and rightmost tape letters, etc.

The channel machine simulates a head transition by cycling through the entire channel once. Moving the head one cell to the right is straightforward, whereas the easiest way to move the head one cell to the left is to use nondeterminism: guess the new head position, and carry on with the simulation only if it is subsequently immediately confirmed that the chosen cell was the right one. All other transitions of the Turing machine are equally straightforward to simulate.

Note that nothing precludes the above procedure from being carried out by a channel machine that suffers from insertion errors; in that case, however, the 'simulation' is not guaranteed to accurately reflect the behaviour of $\mathcal{T}$.

A *space-bounded* computation of $\mathcal{T}$ is one in which $\mathcal{T}$ uses no more than some fixed, predetermined number of tape cells (say n). To simulate such a

computation, one initialises the channel with exactly n blanks, and afterwards strictly alternates read-transitions with write-transitions. In other words, in the absence of insertion errors the channel size remains essentially constant having at all times either n or $n-1$ messages.[4] The simulation proceeds until the channel machine, in attempting to access a blank, is unable to do so. Note that in the presence of insertion errors, the absence of blanks can be ascertained by occurrence-testing for B.

Given $\mathcal{T}$ as above, we construct an ICMET $\mathcal{C} = (S, init, M, C, \Delta)$ composed of several components (cf. Figure 1). One of these components is a 'space-bounded simulator' for $\mathcal{T}$. The simulator has its own dedicated channel, which is at the beginning initialised with a certain number of blanks. (One can ascertain that only blanks are initially on the channel by occurrence-testing for every other message in M.) The simulator then simulates $\mathcal{T}$ until either $\mathcal{T}$ halts, in which case $\mathcal{C}$ also halts, or until all blanks are exhausted, in which case the simulator subroutine returns.

This space-bounded simulator is embedded within a 'decreasing device'. Once all blanks are exhausted and the simulator returns, the decreasing device does the following: it cycles through the whole channel of the simulator and re-initialises every symbol to B (ascertaining success via occurrence-testing). It then deletes one of the Bs and launches a fresh new simulation of $\mathcal{T}$ all over again.

The decreasing device only returns when, upon having re-initialised the simulator's channel with blanks and deleted one, it finds the channel to be empty.

$\mathcal{C}$ also keeps a counter, encoded in unary (using the symbol B, say), which starts at 1 and is subsequently periodically incremented. This counter is at all times stored either on channel *count* or channel *count*$'$. The role of the counter is to indicate how many blanks are to be initially provided upon freshly entering a decreasing-device cycle. This proceeds as follows. Assuming that the counter is stored on channel *count*, for every B in *count* a B is written both onto the simulator's channel and onto *count*$'$. This continues until *count* is empty, at which point control is passed to the space-bounded simulator. (The next time around proceeds similarly except that the roles of *count* and *count*$'$ are inverted, and so on.)

Once the decreasing device returns—upon finding the simulator's channel empty, as explained earlier—$\mathcal{C}$ visits a distinguished control state $t \in S$, increments the counter, and starts a new cycle.

The ICMET $\mathcal{C}$ is represented diagrammatically in Figure 1. Note that, while $\mathcal{T}$ is a deterministic Turing machine, $\mathcal{C}$ is a nondeterministic channel machine.

We claim that $\mathcal{C}$ has an infinite computation that visits control state t infinitely often iff $\mathcal{T}$ does not halt when started on a blank tape. It immediately follows that the recurrent-state problem for ICMETs is undecidable.

It remains to establish the claim. The right-to-left implication is immediate: if $\mathcal{T}$ does not halt, then consider an error-free computation of $\mathcal{C}$. Since by assumption $\mathcal{T}$'s tape contents grows unboundedly with time, every space-bounded

[4] Note that in the 'unconstrained' simulation of $\mathcal{T}$ described earlier, the channel may grow unboundedly (even without insertion errors) as the channel machine periodically adds blanks to it as needed to accurately mimic $\mathcal{T}$'s *unbounded* tape.

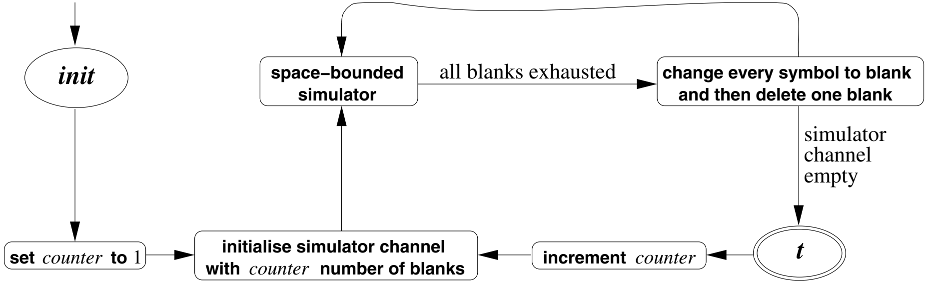


Fig. 1. A schematic representation of the ICMET $\mathcal{C}$. The starting state is *init*, and we are interested in computations that visit state t infinitely often. Note that $\mathcal{C}$ is a nondeterministic machine; the two transitions emerging from 'change every symbol to blank and then delete one blank', for instance, are not mutually exclusive.

simulation of $\mathcal{T}$ eventually exhausts all blanks (recall that $\mathcal{T}$ never writes a fresh blank). As the starting number of blanks in successive space-bounded simulations always decreases by one, the channel always eventually becomes empty and $\mathcal{C}$ therefore always eventually reaches t.

Assume now that $\mathcal{C}$ has an infinite computation that visits t infinitely often, and suppose on the contrary that $\mathcal{T}$ halts. Let n be the total number of tape cells visited by $\mathcal{T}$ in the course of its halting computation. Since $\mathcal{C}$ always increments its counter after visiting t, and since insertion errors can only increase, but not decrease, the value of the counter, eventually the counter reaches some value greater than or equal to n.

At that point, $\mathcal{C}$ initiates a space-bounded simulation of $\mathcal{T}$ starting with p_1 blanks, where $p_1 \geq n$. The simulation continues until no blanks remain on the tape, at which point all symbols are converted to blanks and one blank is deleted. Let us say there are then p_2 blanks on the channel. Note that, although $\mathcal{C}$ never 'knowingly' inserts any extra symbol (blank or otherwise) on the simulator's channel during a space-bounded simulation, insertion errors can occur, so that p_2 could be larger than p_1. In fact, it is clear that $p_2 = p_1 - 1$ iff no insertion error occurred throughout the entire space-bounded simulation (including the channel re-initialisation step).

Continuing in this way, we get a sequence of numbers $p_1, p_2, p_3, \ldots, p_k$ which denote the number of blanks on the channel at the beginning of every space-bounded simulation. Since by assumption the computation of $\mathcal{C}$ we are considering always eventually visits t, and since t can only be reached if the simulator channel is empty, we have $p_k = 0$. Since $p_1 \geq n$, and since the number of blanks decreases by at most 1 in going from any p_i to p_{i+1}, we conclude that there is some j such that $p_j = n$ and $p_{j+1} = n-1$. In other words, the j-th space-bounded simulation was an *error-free* simulation of $\mathcal{T}$ that started on a channel with n blanks. Since $\mathcal{T}$ is deterministic, this simulation should have led to $\mathcal{T}$ halting, which in turn should have halted $\mathcal{C}$ as well, contradicting our initial hypothesis.

This concludes the proof of Theorem 2. □

3 Metric Temporal Logic

We now formally present Metric Temporal Logic (MTL). Given the leisurely background review offered in the Introduction, the present treatment is rather succinct. For a more detailed and comprehensive account of MTL we refer the reader to [6].

A *time sequence* $\tau = \tau_0\tau_1\ldots$ is a finite or infinite sequence of time values $\tau_i \in \mathbb{R}_{\geq 0}$ with $\tau_i \leq \tau_{i+1}$ for all $i < |\tau| - 1$. Here $|\tau|$ denotes the length of τ. If τ is infinite, we require that $\{\tau_i \mid i \in \mathbb{N}\}$ be unbounded (non-Zenoness).

A *timed word* over finite alphabet Σ is a pair $\rho = (\sigma, \tau)$, where $\sigma = \sigma_0\sigma_1\ldots$ is a word over Σ and τ is a time sequence of the same length. We also occasionally refer to a pair (σ_i, τ_i) as a *timed event*, having τ_i as a *timestamp*. Finally, we write $T\Sigma^*$ for the set of finite timed words over alphabet Σ, and $T\Sigma^\omega$ for the set of infinite timed words over Σ.[5]

Given a finite alphabet Σ of atomic events, the formulas of MTL are built up from Σ by Boolean connectives and time-constrained versions of the temporal operators *next* ($\bigcirc$), *eventually* ($\Diamond$), *always* ($\Box$), and *until* ($\mathcal{U}$), as follows:

$$\varphi ::= \top \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid a \mid \bigcirc_I \varphi \mid \Diamond_I \varphi \mid \Box_I \varphi \mid \varphi_1 \,\mathcal{U}_I\, \varphi_2$$

where $a \in \Sigma$, and $I \subseteq \mathbb{R}_{\geq 0}$ is an open, closed, or half-open interval with endpoints in $\mathbb{N} \cup \{\infty\}$. If $I = [0, \infty)$, then we omit the annotation I in the corresponding temporal operator. We also use pseudo-arithmetic expressions to denote intervals. For example, the expression '≥ 1' denotes $[1, \infty)$ and '$=1$' denotes the singleton $\{1\}$.

Given a (finite or infinite) timed word $\rho = (\sigma, \tau)$ and an MTL formula φ, the satisfaction relation $(\rho, i) \models \varphi$ (read ρ satisfies φ at position i) is defined inductively as follows:

- $(\rho, i) \models \top$
- $(\rho, i) \models \varphi_1 \wedge \varphi_2$ iff $(\rho, i) \models \varphi_1$ and $(\rho, i) \models \varphi_2$
- $(\rho, i) \models \neg\varphi$ iff $(\rho, i) \not\models \varphi$
- $(\rho, i) \models a$ iff $i < |\rho|$ and $\sigma_i = a$
- $(\rho, i) \models \bigcirc_I \varphi$ iff $i + 1 < |\rho|$, $(\rho, i + 1) \models \varphi$, and $\tau_{i+1} - \tau_i \in I$
- $(\rho, i) \models \Diamond_I \varphi$ iff there exists j such that $i \leq j < |\rho|$, $(\rho, j) \models \varphi$, and $\tau_j - \tau_i \in I$
- $(\rho, i) \models \Box_I \varphi$ iff for all j such that $i \leq j < |\rho|$, if $\tau_j - \tau_i \in I$ then $(\rho, j) \models \varphi$
- $(\rho, i) \models \varphi_1 \,\mathcal{U}_I\, \varphi_2$ iff there exists j such that $i \leq j < |\rho|$, $(\rho, j) \models \varphi_2$, $\tau_j - \tau_i \in I$, and $(\rho, k) \models \varphi_1$ for all k with $i \leq k < j$.

We say that ρ satisfies φ, denoted $\rho \models \varphi$, if $(\rho, 0) \models \varphi$. Additional Boolean and temporal operators can be defined via the usual conventions. Note that the expected relationships among the constrained temporal operators hold, viz.

[5] Note that we are adopting a *weakly monotonic* view of time, in that several events are allowed to share the same timestamp. The results presented here however carry over verbatim under a strongly monotonic interpretation of time.

$\Diamond_I \varphi \equiv \top \; \mathcal{U}_I \; \varphi$ and $\Box_I \varphi \equiv \neg \Diamond_I \neg \varphi$. We have nonetheless defined $\Diamond_I$ and $\Box_I$ separately because our main undecidability result does not require the $\mathcal{U}_I$ operators.

Büchi timed automata [3] are real-time extensions of Büchi automata that accept infinite timed words. It is not necessary for our purposes to say anything more about these, other than to state that there exists a (rather trivial) Büchi timed automaton that accepts $T\Sigma^\omega$, the set of all infinite timed words.

Given an MTL formula φ, the *finite-satisfiability problem* asks if there exists a finite timed word that satisfies φ; this problem was shown to be decidable, with non-primitive recursive complexity, in [19]. The *infinite-satisfiability problem* asks if there is an infinite timed word that satisfies φ. Finally, the *infinite model-checking problem* asks, given a Büchi timed automaton $\mathcal{A}$, whether all infinite timed words accepted by $\mathcal{A}$ satisfy φ. The main result of this paper is the following:

Theorem 3. *The infinite-satisfiability and infinite model-checking problems for MTL are undecidable. In fact, these problems are already undecidable for the fragment of MTL that excludes all constrained 'until' operators $\mathcal{U}_I$.*

Proof. The infinite-satisfiability part follows immediately from Theorem 4.3 (in the next section) and Theorem 2.

For the infinite model-checking statement, consider a universal Büchi timed automaton (i.e., one that accepts every timed word). Model checking this automaton against an MTL formula is equivalent to asking whether the formula is valid, i.e., whether its negation is unsatisfiable. □

4 Two-Way Reductions

We exhibit a correspondence between the faulty Turing machines studied in Section 2 and Metric Temporal Logic formulas. More precisely, we show how to effectively translate finite (respectively infinite) MTL satisfiability questions into halting (respectively recurrent-state) problems for ICMETs, and vice-versa.

The advantage of this correspondence is that many questions about MTL, whose dense-time semantics is sometimes considered somewhat awkward and counter-intuitive [15], can be translated into the purely discrete framework of ICMETs.

Theorem 4. *The following reductions between ICMETs and MTL formulas are all effective:*

1. *Let $(\mathcal{C}, t)$ be an instance of the halting problem for ICMETs. Then there exists an MTL formula φ such that $\mathcal{C}$ reaches t iff φ is satisfiable by some finite word.*
2. *Let φ be an MTL formula. Then there exists an ICMET $\mathcal{C}$ together with a distinguished control state t of $\mathcal{C}$ such that φ is satisfiable by some finite word iff $\mathcal{C}$ reaches t.*

3. *Let $(\mathcal{C}, t)$ be an instance of the recurrent-state problem for ICMETs. Then there exists an MTL formula φ such that $\mathcal{C}$ has a t-recurrent computation iff φ is satisfiable by some infinite word.*
4. *Let φ be an MTL formula. Then there exists an ICMET $\mathcal{C}$ together with a distinguished control state t of $\mathcal{C}$ such that φ is satisfiable by some infinite word iff $\mathcal{C}$ has a t-recurrent computation.*

Moreover, for statements 1 and 3 the fragment of MTL that excludes all constrained 'until' operators $\mathcal{U}_I$ suffices.

Proof. For the purposes of this paper the most important statement is 3, and accordingly we give full details of that proof and briefly comment on the other cases afterwards.

Let $\mathcal{C} = (S, init, M, C, \Delta)$ be an ICMET, with $t \in S$ the distinguished control state. The idea is to encode valid t-recurrent computations of $\mathcal{C}$ as timed words, that are in turn captured by an MTL formula φ.

To this end, assume that $\mathcal{C}$ has k channels, say $C = \{c_1, \dots, c_k\}$. Define an alphabet $\Sigma = S \cup M \cup \Delta \cup \{B_i, E_i \mid 1 \leq i \leq k\}$. A global state $(s, x_1, \dots, x_k)$ of $\mathcal{C}$ is encoded as a finite timed word of total duration $2k$, as follows:

- s occurs at time 0.
- The contents x_i of channel c_i is encoded in the open interval $(2i - 1, 2i)$ as a matching sequence of timed events. The latest event corresponds to the message at the head of the channel, and so on.
- The event B_i occurs at time $2i - 1$, and the event E_i occurs at time $2i$. These two events therefore delineate the contents of channel c_i.

Moreover, the timed word is strongly monotonic (no two timed events share the same timestamp), and contains no timed events other than the ones listed above. In particular, the open time intervals $(2i, 2i + 1)$ are empty.

Note that the density of time allows such timed words to accommodate arbitrarily large channel contents. Note also that any such timed word can be uniquely converted into a global state of $\mathcal{C}$.

A computation $(s_0, \overline{x_0}) \xrightarrow{\alpha_0} (s_1, \overline{x_1}) \xrightarrow{\alpha_1} (s_2, \overline{x_2}) \xrightarrow{\alpha_2} \cdots$ of $\mathcal{C}$ can then be encoded as an infinite timed word, by time-shifting and concatenating the timed words corresponding to each global state and interspersing the transitions α_j, as follow:

- If s_0 occurs at time τ_0, then s_j occurs at time $(2k + 2)j + \tau_0$, followed by the encoding of the remainder of the j-th global state, as detailed above.
- α_j occurs at time $(2k + 2)(j + 1) - 1 + \tau_0$, i.e., exactly one time unit after the end of the encoding of the j-th global state, and exactly one time unit before event s_{j+1}.
- If message m in channel c_i, in global state j, is not read off while performing the transition α_j, then the difference between the timestamps of the two occurrences of m in the encodings of the $(j+1)$-th and j-th global states is exactly $2k + 2$.

The last clause ensures that channel contents are preserved between transitions; nothing prevents, however, insertion errors from occurring, in the form of timed events with no matching events $2k+2$ time units *earlier*.

Observe that any infinite computation of $\mathcal{C}$ can immediately be recovered from its encoding as an infinite timed word.

It remains to exhibit an MTL formula φ that captures precisely the timed words corresponding to the t-recurrent infinite computations of $\mathcal{C}$. We first build various useful components, as follows:

We first want to restrict ourselves to strongly monotonic timed words:

$$\varphi_{\mathsf{sm}} = \Box \bigcirc_{>0} \top .$$

The first event is the control state *init*, and afterwards control states are forever spaced exactly $2k+2$ time units apart:

$$\varphi_S = \mathit{init} \wedge \Box \left(\bigvee S \implies \left(\Diamond_{=2k+2} \bigvee S \wedge \Box_{<2k+2} \neg \bigvee S \right) \right) .$$

The structure of global-state encodings is captured by the following formulas, for $1 \leq i \leq k$:

$$\begin{aligned}
\varphi_{B_i} &= \Box \left(\bigvee S \implies \left(\Diamond_{=2i-1} B_i \wedge \Box_{[0,2i-1)\cup(2i-1,2k+2)} \neg B_i \right) \right) \\
\varphi_{E_i} &= \Box \left(\bigvee S \implies \left(\Diamond_{=2i} E_i \wedge \Box_{[0,2i)\cup(2i,2k+2)} \neg E_i \right) \right) \\
\varphi_{c_i} &= \Box \left(\bigvee S \implies \left(\Box_{(2i-1,2i)} \bigvee M \wedge \Box_{(2i,2i+1)} \bot \right) \right) .
\end{aligned}$$

Interspersing transitions:

$$\varphi_\Delta = \Box \left(\bigvee S \implies \left(\Diamond_{=2k+1} \bigvee \Delta \wedge \Box_{<2k+1} \neg \bigvee \Delta \wedge \Box_{(2k+1,2k+2)} \bot \right) \right) .$$

We now define components that ensure the validity of the encoded computation.

Consecutive control states should match the source and target of the intervening transitions; to this end, for any pair of control states s, s', let $\Delta_{s,s'} = \{(s, -, s') \in \Delta\}$.

$$\varphi_{\Delta S} = \bigwedge_{s,s' \in S} \Box \left((s \wedge \Diamond_{=2k+2} s') \implies \Diamond_{=2k+1} \bigvee \Delta_{s,s'} \right) .$$

To handle channel integrity, first define:

$$\varphi_{\mathsf{copy}} = \Box_{(0,1)} \bigwedge_{m \in M} (m \implies \Diamond_{=2k+2} m) .$$

Then, for $1 \leq i \leq k$ and $m \in M$, let:

$$\varphi_{c_i=\emptyset} = \bigvee S \wedge \bigwedge_{1\leq j\leq k} \Diamond_{=2j-1}\varphi_{\mathsf{copy}} \wedge \Box_{(2i-1,2i)}\bot$$

$$\varphi_{c_i!m} = \bigvee S \wedge \bigwedge_{1\leq j\leq k} \Diamond_{=2j-1}\varphi_{\mathsf{copy}} \wedge \Diamond_{[2i-1,2i)} (\bigcirc E_i \wedge \Diamond_{=2k+2} \bigcirc m)$$

$$\varphi_{c_i?m} = \bigvee S \wedge \bigwedge_{\substack{1\leq j\leq k \\ j\neq i}} \Diamond_{=2j-1}\varphi_{\mathsf{copy}} \wedge \Diamond_{=2i-1} \bigcirc (m \wedge \varphi_{\mathsf{copy}})\,.$$

Channel contents should vary according to the relevant transitions. Recall that $L = \{c!m, c?m, c{=}\emptyset \mid c \in C \wedge m \in M\}$. For $l \in L$, let $\Delta_l = \{(-, l, -) \in \Delta\}$.

$$\varphi_{\Delta C} = \Box\left(\bigvee S \implies \bigwedge_{l\in L}\left(\Diamond_{=2k+1}\bigvee \Delta_l \implies \varphi_l\right)\right),$$

where the formulas φ_l are defined above.

We are interested in t-recurrent computations of $\mathcal{C}$, which are captured by requiring:

$$\varphi_{\mathsf{rec}} = \Box\Diamond t.$$

Finally, let:

$$\varphi = \varphi_{\mathsf{sm}} \wedge \varphi_S \wedge \bigwedge_{1\leq i\leq k} (\varphi_{B_i} \wedge \varphi_{E_i} \wedge \varphi_{c_i}) \wedge \varphi_\Delta \wedge \varphi_{\Delta S} \wedge \varphi_{\Delta C} \wedge \varphi_{\mathsf{rec}}.$$

By construction, infinite timed words that satisfy φ can be translated into valid t-recurrent computations of $\mathcal{C}$, and vice-versa. It is also clear that φ does not use any $\mathcal{U}_I$ operator, concluding the proof of Statement 3.

Note that a proof of Statement 1 can easily be engineered along the same lines as the above.

For Statement 4, one first reduces infinite satisfiability for MTL to a non-emptiness problem for one-clock timed alternating automata with a weak parity acceptance condition, by extending the construction presented in [19]. Next, one translates this non-emptiness problem into the existence of a Büchi path in a certain well-structured transition system, which can itself be described using a perfect channel machine, again following a construction of [19]. One then argues that insertion errors can only cause valid Büchi paths to be rejected, thereby preserving correctness.

Finally, Statement 2 can be handled by following a simplified version of the above procedure. $\Box$

5 Summary

The main result of this paper is that the satisfiability and model checking problems for Metric Temporal Logic, interpreted over infinite timed words, are

MTL	**ICMET**	Complexity
Finite satisfiability	Halting problem	Non-primitive recursive
Infinite satisfiability	Recurrent-state problem	Undecidable

Fig. 2. A summary of the two-way reductions between Metric Temporal Logic and faulty Turing machine problems

undecidable. As such, this closes a gap between a host of decidability and undecidability results for various variants of MTL. The crux of our approach is to establish a strong correspondence between problems about Metric Temporal Logic and problems about ICMETs, a particular brand of faulty Turing machines, as depicted in Figure 2.

An interesting question is whether this correspondence can be leveraged, in one direction or the other, to obtain additional results or insights about the two entities MTL and ICMET.

References

[1] P. Abdulla and B. Jonsson. Undecidable verification problems with unreliable channels. *Inf. Comput.*, 130:71–90, 1996.

[2] P. A. Abdulla, J. Deneux, J. Ouaknine, and J. Worrell. Decidability and complexity results for timed automata via channel machines. In *Proc. ICALP*, volume 3580 of *Springer LNCS*, 2005.

[3] R. Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126: 183–235, 1994.

[4] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43:116–146, 1996.

[5] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Proc. RTTP*, volume 600 of *Springer LNCS*, 1992.

[6] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. *Inf. Comput.*, 104:35–77, 1993.

[7] R. Alur and T. A. Henzinger. A really temporal logic. *J. ACM*, 41:181–204, 1994.

[8] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[9] G. Cécé, A. Finkel, and S. P. Iyer. Unreliable channels are easier to verify than perfect channels. *Inf. Comput.*, 124:20–31, 1996.

[10] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[11] T. A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems.* PhD thesis, Stanford University, 1991. Tech. rep. STAN-CS-91-1380.

[12] T. A. Henzinger. It's about time: Real-time logics reviewed. In *Proc. CONCUR*, volume 1466 of *Springer LNCS*, 1998.

[13] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proc. ICALP*, volume 623 of *Springer LNCS*, 1992.

[14] T. A. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. The regular real-time languages. In *Proc. ICALP*, volume 1443 of *Springer LNCS*, 1998.

[15] Y. Hirshfeld and A. Rabinovich. Logics for real time: Decidability and complexity. *Fundam. Inform.*, 62(1):1–28, 2004.

[16] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time Systems*, 2(4):255–299, 1990.
[17] S. Lasota and I. Walukiewicz. Personal communication, 2005.
[18] S. Lasota and I. Walukiewicz. Alternating timed automata. In *Proc. FOSSACS*, volume 3441 of *Springer LNCS*, 2005.
[19] J. Ouaknine and J. Worrell. On the decidability of metric temporal logic. In *Proc. LICS*. IEEE Press, 2005.
[20] J. Ouaknine and J. Worrell. Safety metric temporal logic is fully decidable. Submitted, 2005.
[21] J.-F. Raskin and P.-Y. Schobbens. State-clock logic: A decidable real-time logic. In *Proc. HART*, volume 1201 of *Springer LNCS*, 1997.
[22] K. Schneider. *Verification of Reactive Systems*. Springer, 1997.
[23] P. Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Inf. Process. Lett.*, 83(5):251–261, 2002.
[24] T. Wilke. Specifying timed state sequences in powerful decidable logics and timed automata. In *Proc. FTRTFTS*, volume 863 of *Springer LNCS*, 1994.

Denotational Semantics of Hybrid Automata*

Abbas Edalat[1] and Dirk Pattinson[2]

[1] Department of Computing, Imperial College London, UK
[2] Department of Computer Science, University of Leicester, UK

Abstract. We introduce a denotational semantics for non-linear hybrid automata, and relate it to the operational semantics given in terms of hybrid trajectories. The semantics is defined as least fixpoint of an operator on the continuous domain of functions of time that take values in the lattice of compact subsets of n-dimensional Euclidean space. The semantic function assigns to every point in time the set of states the automaton can visit at that time, starting from one of its initial states. Our main results are the correctness and computational adequacy of the denotational semantics with respect to the operational semantics and the fact that the denotational semantics is computable.

1 Introduction

A hybrid automaton [12, 2] is a digital, real-time system that interacts with an analogue environment. Hybrid automata are ubiquitous in all areas of modern engineering and technology. For example, the (digital) height control of an automobile chassis depends on and influences the (continuous) driving conditions of the vehicle [18]. Hybrid automata typically operate in safety critical areas, such as the highway control systems [17] and air traffic control [20]. They combine a finite set of control states with continuous dynamics. In every control state, the continuous variables evolve according to an ordinary differential equation and the system changes control states if the continuous variables reach certain thresholds.

One of the key concerns in the theory of hybrid automata is the algorithmic verification of safety critical properties. This problem is well understood for linear systems [3] and implemented in the model checker HyTech [13]. The situation for non-linear systems is, not surprisingly, much less satisfactory. While the approximation of non-linear hybrid automata by linear systems is asymptotically complete [14], it results in a huge blow-up in the number of discrete control states and associated state transitions, which limits the possibilities of algorithmic analysis.

This paper presents an alternative approach. Conceptually, we regard a hybrid automaton as the integration of two different types of systems: the evolution of a family of continuous systems, governed by differential equations, and the dynamics of a discrete system given by a generalised iterated function system (IFS), see [16]. We synthesise the domain-theoretic approach to solving differential equations [7, 10] and the domain-theoretic approach to obtain the attractor of an iterated function system [6] to develop a domain-theoretic semantics for general hybrid automata. The denotational semantics assigns to every time point t the set $[\![H]\!](t)$ of states that the automaton H

* This work has been partially supported by DFG (Germany) and the European Union.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 231–245, 2006.
© Springer-Verlag Berlin Heidelberg 2006

can enter at time t. The semantic function $[\![H]\!]$ is obtained as the least fixpoint in the (continuous) domain of compact-set valued functions of a real variable. Our first main results are correctness and computational adequacy of this denotational semantics w.r.t. the operational semantics, given in terms of a labelled transition system. Moreover, standard techniques of domain theory allow us to actually compute this function. The implications are twofold: we obtain new results on the computability of trajectories in the domain theoretic model, and our analysis gives rise to a directly implementable algorithm that computes approximations to the semantic function $[\![H]\!]$ up to an arbitrary degree of accuracy. As the algorithm works on proper data types, defined e.g. over the dyadic numbers, this property is moreover guaranteed for implementations.

The paper is divided in two parts. In the first part, we focus on flow automata, where the behaviour of the continuous variables in every discrete control state is governed by flow functions, which behave like the solutions of ordinary differential equations. We impose two conditions on the automata under scrutiny: first, we require that the ingredients of the automaton give rise to Scott continuous functions on the respective domains. In order to show that the least fixpoint precisely captures the reachable states, we assume that the automaton is separated, i.e. has no transient states which the automaton can leave immediately (after 0 time units) after entering. We discuss these restrictions by means of examples, and show that the semantic function associated with a flow automaton cannot be computable in absence of these properties.

In the second part of the paper, we transfer the results obtained to hybrid automata, where the trajectories of the continuous variables are given by a vector field. By instantiating earlier results on domain theoretic solutions of initial value problems, we reduce the problem of computing the semantic function of a hybrid automaton to that of a flow automaton. Taken together, the domain theoretic approach provides a new computational model for the analysis of hybrid systems, and gives rise to both new computability results, and directly implementable data types and algorithms for the analysis of non-linear systems.

Related Work. We have already mentioned symbolic techniques for the analysis of linear hybrid automata [3] and their implementation in the HyTech model checker [13]. The domain theoretic approach of this paper is related to the interval analysis approach of [15], where interval numerical methods are used to compute over-approximations of the set of reachable states. In contrast to *loc.cit.*, where outward rounding is required if the result of an arithmetic operation is not machine representable, the domain theoretic model of computation actually allows to compute the semantic function up to an arbitrary degree of accuracy.

2 Preliminaries and Notation

We use basic domain theoretic notions, see e.g. [1, 11]. In particular, our analysis employs the following domains defined over the real numbers: the domain of n-dimensional compact rectangles extended with a least element

$$\mathbf{I}\mathbb{R}^n = \{a \subseteq \mathbb{R}^n \mid a \text{ nonempty compact rectangle}\} \cup \{\mathbb{R}^n\},$$

ordered by reverse inclusion, and the *extended upper space*

$$\mathbf{U}^{\top}\mathbb{R}^n = \{c \subseteq \mathbb{R}^n \mid c \text{ compact}\} \cup \{\mathbb{R}^n\}$$

of compact subsets of $\mathbb{R}^n$, also ordered by reverse inclusion. Note that the extended upper space arises by extending the upper space [5] with the top element $\top = \emptyset$. A closed *semi rectangle* in $\mathbb{R}^n$ is of the form $a_1 \times \cdots \times a_n$, where the a_i are closed (not necessarily bounded) intervals in $\mathbb{R}$. If A is a semi-rectangle, we write $\mathbf{I}\mathsf{A} = \{\mathsf{A} \cap r \mid r \in \mathbf{I}\mathbb{R}^n\}$ and $\mathbf{U}^{\top}\mathsf{A} = \{\mathsf{A} \cap c \mid c \in \mathbf{U}^{\top}\mathbb{R}^n\}$ for the sub-domain of all elements above A. In particular, we will consider the domain $\mathbf{I}[0, \infty)$, whose bottom element is $\perp = [0, \infty)$. For a semi rectangle A, $\mathbf{I}\mathsf{A}$ is a continuous Scott domain and $\mathbf{U}^{\top}\mathsf{A}$ is a continuous lattice. We often consider $\mathbf{I}\mathsf{A} \subseteq \mathbf{U}^{\top}\mathsf{A}$ as a sub-domain without making this explicit; similarly, we identify $x \in \mathbb{R}^n$ with the degenerate hyper-rectangle $\{x\} \in \mathbf{I}\mathbb{R}^n \subseteq \mathbf{U}^{\top}\mathbb{R}^n$. We write $\perp = \mathsf{A}$ for the least element of both $\mathbf{I}\mathsf{A}$ and $\mathbf{U}^{\top}\mathsf{A}$, and $\top = \emptyset$ for the top element of $\mathbf{U}^{\top}\mathsf{A}$. Note that the way-below relation, both in $\mathbf{I}\mathsf{A}$ and $\mathbf{U}^{\top}\mathsf{A}$, is given by $a \ll b$ iff $b \subseteq a^o$, where a^o is the interior of a.

If $(C_i)_{i \in I}$ is a family of compact subsets $C_i \subseteq \mathbb{R}^{n_i}$, we identify $(x_i)_{i \in I} \in \prod_{i \in I} \mathbf{U}^{\top} C_i$ with the set $\{(i, y) \mid i \in I, y \in x_i\}$ for convenience of notation. Note that this induces a membership predicate and subset relation, which are explicitely given by

$$(j, z) \in (x_i)_{i \in I} \iff z \in x_j \text{ and } (x_i)_{i \in I} \subseteq (y_i)_{i \in I} \iff \forall i \in I. x_i \subseteq y_i$$

where $(x_i)_{i \in I}$ and $(y_i)_{i \in I} \in \prod_{i \in I} \mathbf{U}^{\top} C_i$, $j \in I$ and $z \in C_j$. Moreover, we obtain two continuous maps $\bigcap, \bigcup$, whose explicit definition reads

$$\diamond : (\prod_{i \in I} \mathbf{U}^{\top} C_i)^2 \to \prod_{i \in I} \mathbf{U}^{\top} C_k, \quad ((x_i)_{i \in I}, (y_i)_{i \in I}) \mapsto (x_i \diamond y_i)_{i \in I}$$

where $\diamond \in \{\bigcap, \bigcup\}$. Note that, domain theoretically, $\bigcap$ is the least upper bound and $\bigcup$ gives us the greatest lower bound of two elements of $\prod_{i \in I} \mathbf{U}^{\top} C_i$. We always consider sub-domains of the upper space or the interval domain as equipped with the Scott topology.

The symbol $\Rightarrow$ is used for the continuous function space. In particular, for semi rectangles A, B, we consider the set $(\mathsf{A} \Rightarrow \mathbf{U}^{\top}\mathsf{B})$ of functions $f : \mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}$ which are continuous with respect to the Euclidean topology on A and the Scott topology on $\mathbf{U}^{\top}\mathsf{B}$. Similarly, $(\mathbf{U}^{\top}\mathsf{A} \Rightarrow \mathbf{U}^{\top}\mathsf{B})$ denotes the set of functions that are continuous w.r.t. the Scott topology on $\mathbf{U}^{\top}\mathsf{A}$ and $\mathbf{U}^{\top}\mathsf{B}$; the same applies to the interval domain.

We extend the ordinary arithmetical operations to the upper space without further mention. In particular, we write $a \diamond b = \{x \diamond y \mid x \in a, y \in b\}$, where $\diamond \in \{+, -, *, /\}$ and $a, b \in \mathbf{U}^{\top}\mathbb{R}^n$. (We adopt the standard convention that $a/b = \perp$ if $0 \in b$.)

It is a straightforward exercise to see that Scott continuous functions of type $\mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}$ are precisely the semi continuous functions of set-valued analysis [4]. More concretely, we have that $f : \mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}$ is Scott continuous, iff

$$\forall x \in A \forall \epsilon > 0 \exists \delta > 0 \forall x' \in B_\epsilon(x). f(x') \subseteq f(x) + B_\delta$$

where $B_\epsilon(x) = \{x' \in \mathsf{A} \mid \|x - x'\| < \epsilon\}$ and $B_\delta = B_\delta(0)$. Note that we have the Scott continuous *extension mapping*

$$\mathcal{E} : (A \Rightarrow \mathbf{U}^{\top}\mathsf{B}) \to (\mathbf{U}^{\top}\mathsf{A} \Rightarrow \mathbf{U}^{\top}\mathsf{B}), f \mapsto \lambda x. \bigcap_{y \in x} f(y),$$

and it is an easy exercise to show that this greatest lower bound is actually given by direct image, i.e. $\mathcal{E}(f)(x) = \bigcup\{f(y) \mid y \in x\}$.

3 Flows and Flow Automata

We begin our study of hybrid automata by first discussing *flow automata*, where the continuous evolution in every control state is an explicitly given flow function. This will subsequently be shown to be equivalent to the case that the continuous evolution is specified by a vector field in Section 6. For flow automata, every discrete control state comes with a flow function that behaves like the solution of an initial value problem, and governs the evolution of the continuous variables in that state.

Definition 1. *A* flow *is a continuous function $f : \mathbb{R}^n \times [0, \infty) \to \mathbb{R}^n$, which is continuously differentiable w.r.t. its last argument, such that $f(x, 0) = x$ and $f(x, s + t) = f(f(x, s), t)$ for all $x \in \mathbb{R}^n$ and $s, t \in [0, \infty)$.*

That is, a flow $f : \mathbb{R}^n \times [0, \infty) \to \mathbb{R}^n$ behaves like the solution of an initial value problem $\dot{f}(t) = v(f(t)), f(0) = x$, where v is defined on the whole of Euclidean space $\mathbb{R}^n$. Note that flows typically arise as solutions of initial value problems:

Lemma 1. *Suppose $v : \mathbb{R}^n \to \mathbb{R}^n$ is a Lipschitz vector field. If $f(x, \cdot)$ denotes the (unique) solution of the initial value problem $f(x, t) = v(f(x, t)), f(x, 0) = x$, then f is a flow. We say that f is the* flow induced by v.

We now introduce continuous flow automata.

Definition 2. *A* flow automaton *in $\mathbb{R}^n$ is a tuple $F = (Q, \mathrm{inv}, \mathrm{flow}, \mathrm{res}, \mathrm{init})$ where*

- *Q is a finite set of discrete control states*
- $\mathrm{inv} = (\mathrm{inv}(q))_{q \in Q}$ *is a family of* state invariants *with* $\mathrm{inv}(q) \subseteq \mathbb{R}^n$
- $\mathrm{flow} = (\mathrm{flow}(q))_{q \in Q}$ *is a family of* flow functions $\mathrm{flow}(q) : \mathbb{R}^n \times [0, \infty) \to \mathbb{R}^n$
- $\mathrm{res} = (\mathrm{res}(p, q))_{p,q \in Q}$ *is a family of* reset relations *with* $\mathrm{res}(p, q) : \mathrm{inv}(p) \to \mathcal{P}(\mathrm{inv}(q))$
- $\mathrm{init} = (\mathrm{init}(q))_{q \in Q}$ *is a family of* initial states *with* $\mathrm{init}(q) \subseteq \mathrm{inv}(q)$.

We call a flow automaton compact, *if* $\mathrm{inv}(q), \mathrm{init}(q) \in \mathbf{U}^{\top}\mathbb{R}^n$ *are compact for all $q \in Q$ and* $\mathrm{res}(p, q)(x) \in \mathbf{U}^{\top}\mathrm{inv}(q)$ *is a compact subset of* $\mathrm{inv}(q)$ *for all $p, q \in Q$ and all $x \in \mathrm{inv}(q)$. A* state *of a flow automaton is a tuple (q, x) with $q \in Q$ and $x \in \mathrm{inv}(q)$. We write $S_F = \{(q, x) \mid q \in Q, x \in \mathrm{inv}(q)\}$ for the state space of F and $i_F = \{(q, x) \in S \mid x \in \mathrm{init}(q)\}$ for the set of initial states.*

Although our interest in the flow function will be restricted to values $\mathrm{flow}(q)(x, t)$, where $x \in \mathrm{inv}(q)$, the flow function is defined on the whole of Euclidean space for convenience.

The above definition of flow automata, while slightly different, is equivalent to the standard definition given e.g. in [3]. While our control states are in one-to-one correspondence to the control locations of *loc.cit.*, the transitions between control states are

modelled in terms of a finite multiset $V \subseteq Q \times Q$ of transitions, and an action predicate $\mathrm{act}(v) \subseteq \mathbb{R}^n \times \mathbb{R}^n$ is assigned to every transition $v \in V$. In this terminology, the automaton can change its state, say from state (q, x) to state (q', x') iff there exists a transition $(q, q') \in V$ with $(x, x') \in \mathrm{act}(v)$. In our terminology, this can be modelled by the reset relation $\mathrm{res}(q, q') = \lambda x.\{y \in \mathrm{inv}(q) \mid \exists (q, q') \in V.(x, y) \in \mathrm{act}(q, q')\}$.

For the remainder of the paper, we assume that all flow automata are compact. Our main interest lies in the comparison of the denotational semantics and the operational semantics of a flow automaton. The latter is given in terms of a labelled transition system, where a label is either a non-negative real numbers, that signifies time, or τ, indicating that the automaton is changing its discrete control state.

Definition 3. *Suppose* $F = (Q, \mathrm{inv}, \mathrm{flow}, \mathrm{res}, \mathrm{init})$ *is a flow automaton and let* $\Sigma = [0, \infty) \cup \{\tau\}$. *The* associated transition system T_F *is the tuple* $(S_F, \rightarrow)$*, where* S_F *is the state space of* F *and* $\rightarrow \subseteq S \times \Sigma \times S$ *is defined by the following two clauses:*

flow transitions $(q, x) \rightarrow^t (q', x')$ *iff* $q = q'$, $\mathrm{flow}(q)(x, t_0) \in \mathrm{inv}(q)$ *for all* $t_0 \in [0, t]$ *and* $\mathrm{flow}(q)(x, t) = x'$
jump transitions $(q, x) \rightarrow^\tau (q', x')$ *iff* $x' \in \mathrm{res}(q, q')(x)$

For states $s, s' \in S$*, we write* $s \rightarrow_*^t s'$ *if there is a finite sequence of states* $s_1, \ldots, s_k$ *with* $s \rightarrow^{\delta_1} s_1 \rightarrow^{\delta_2} \cdots \rightarrow^{\delta_k} s_k = s'$ *with* $\delta_1, \ldots, \delta_k \in \Sigma$ *and* $\sum_{\delta_k \in [0,\infty)} \delta_k = t$*. We write* $\mathrm{init} \rightarrow_*^t s$ *iff there exists* $i \in i_F$ *with* $i \rightarrow_*^t s$*.*

An F-trajectory *is a finite or infinite sequence* $(t_i, q_i, f_i)_{i<N}$ *where* $N \in \mathbb{N} \cup \{\infty\}$ *such that* $(t_i)_{i<N}$ *is non-decreasing in* $[0, \infty)$*,* $(q_i)_{i<N}$ *is a sequence in* Q *and* $f_i : [t_{i-1}, t_i] \rightarrow \mathbb{R}^n$ *is a function (we use the convention that* $t_{-1} = 0$*) that, for all* $i < N$*, satisfies*

- $f_0(t_{-1}) \in \mathrm{init}(q_0)$ *and* $(q_i, f_i(t_{i-1})) \rightarrow^t (q_i, f_i(t_{i-1} + t))$ *for all* $t \in [t_{i-1}, t_i]$
- $(q_i, f_i(t_i)) \rightarrow^\tau (q_{i+1}, f_{i+1}(t_i))$.

We denote the set of possible states of the automaton F *at time* t *by* $R_F(t)$ *and the set of all states the automaton can visit up to time* t *by* $V_F(t)$*, formally defined by*

$$R_F(t) = \{s \in S_F \mid \mathrm{init} \rightarrow_*^t s\} \quad \textit{and} \quad V_F(t) = \bigcup \{R_F(s) \mid s \leq t\}$$

where $t \in [0, \infty)$.

Note that by assumption, $\mathrm{flow}(q_i)(f_i(t_{i-1}), t) = f_i(t_{i-1} + t)$. Compared with the definition of trajectories in [2], it is straightforward to verify that, under the correspondence outlined after Definition 2, our definition of trajectories gives rise to the same semantics.

We now turn to the main issue of the present paper and describe the necessary ingredients needed to perform domain theoretic analysis of a flow automaton F. Our main goal is to define a domain theoretic semantic function $[\![F]\!] : [0, \infty) \rightarrow \prod_{q \in Q} \mathbf{U}^\top \mathrm{inv}(q)$. The function $[\![F]\!]$ associated to every time point $t \in [0, \infty)$ an element of $\prod_{q \in Q} \mathbf{U}^\top \mathrm{inv}(q)$. That is, to every point in time t we associate a family $(s_q)_{q \in Q}$, with $s_q \subseteq \mathrm{inv}(q)$, of compact sets such that $\{(q, x) \mid x \in s_q\} = R_F(t)$. Having computed R_F, it is easy to derive a mechanism for computing the possibly visited states $V_F(t)$ at time t by unfolding the definition of V_F. We demonstrate later, that it is also possible to obtain V_F directly as a fixed point.

The goal of the construction is to give a *continuous* semantics of flow automata: if the automaton is *effectively given*, i.e. both flow and res arise as limits of sequences of finitary approximations with $\text{flow} = \bigsqcup_{k\in\mathbb{N}} f_k$ and $\text{res} = \bigsqcup_{k\in\mathbb{N}} r_k$, then we can effectively obtain $\sigma_k : [0, \infty) \to \prod_{q\in Q} \text{inv}(q)$ such that $[\![F]\!] = \bigsqcup_{k\in\mathbb{N}} \sigma_k$. This provides us with three important properties:

1. Every σ_k is a *conservative approximation* of the semantics of F, for $k \geq 0$.
2. The semantics of F can be computed up to an arbitrary degree of accuracy.
3. The algorithm for computing σ_k can be implemented on a digital computer without loss of precision.

Clearly, continuity of the semantics mapping $[\![\cdot]\!]$ can only be achieved if we restrict attention to flow automata, whose components are continuous. This motivates the next definition.

Definition 4. *A flow automaton* $F = (Q, \text{inv}, \text{flow}, \text{res}, \text{inv})$ *is* continuous*, if* $\text{res}(p, q) : \text{inv}(p) \to \mathbf{U}^\top \text{inv}(q)$ *is Scott continuous for all* $p, q \in Q$*. We say that* F *is* separated*, if*

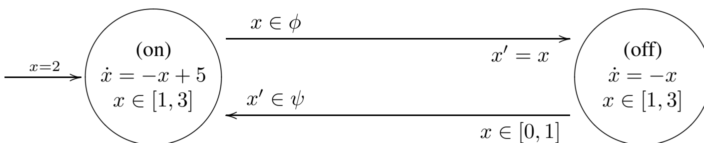
- $x \in \text{res}(p, q)(y)$ *implies that* $\text{res}(q, r)(x) = \emptyset$ *for all* $p, q, r \in Q$ *and* $y \in \text{inv}(p)$
- $x \in \text{init}(q)$ *implies that* $\text{res}(q, r)(x) = \emptyset$ *for all* $q, r \in Q$

While the continuity condition on res is clearly enforced by our goal to be able to approximate the semantics of flow automata, the separation condition tells us that there are no transient states, i.e. the automaton cannot perform state changes from q_0 to q_1, and subsequently from q_1 to q_2 without remaining in state q_1 for a non-zero amount of time.

We will see later that separation and continuity imply that the automaton under scrutiny is non-zeno. While we believe that all of our results can be established even for non-separated automata under the additional assumption that the automata are non-zeno, the main benefit of the separation property is that it is very easy to verify.

For a continuous flow automaton, the family $\text{res}(p, q)_{p,q\in Q}$ induces a generalised IFS on the extended upper spaces of $\text{inv}(p)$, for $p \in Q$, as we will see in Definition 5 later on. The following example discusses the requirements introduced in Definition 4.

Example 1. We consider the following variant F of a thermostat automaton, see e.g. [14]. Let $Q = \{\text{on}, \text{off}\}$ with $\text{inv}(q) = [1, 3]$ for $q = \text{on}, \text{off}$. The flow functions are given by the differential equations $\text{flow}(\text{on})(x_0, \cdot) =$ the unique solution of $\dot{x} = -x + 5, x(0) = x_0$, and similarly, $\text{flow}(\text{off})(x_0, \cdot) =$ the unique solution of $\dot{x} = -x, x(0) = x_0$, with initial state $(\text{on}, 2)$. We fix two subsets $\phi, \psi \subseteq [1, 3]$ and let $\text{res}(\text{on}, \text{off})(x) = \{x\} \cap \psi$. The function $\text{res}(\text{off}, \text{on})$ is given by $x \mapsto \psi$, if $x \in [0, 1]$, and $x \mapsto \emptyset$ otherwise. Graphically, the automaton can be displayed as follows, where x' denotes the value of x after the change of control states.



We now discuss several alternatives for the sets ϕ and ψ, and relate them to continuity of the induced automaton.

1. Suppose $\psi = (1,2)$. Then $\mathrm{res}(\mathrm{off},\mathrm{on})$ does not take values in $\mathbf{U}^{\top}[1,3]$, as $(1,2)$ is not compact, hence $\mathrm{res}(\mathrm{off},\mathrm{on})$ is not a well defined function of type $[1,3] \to \mathbf{U}^{\top}[1,3]$.
2. Suppose $\phi = (2,3]$. Then the F is not continuous, as for $x = 2$ and $\epsilon > 0$, we fail to find δ s.t. for all $x' \in B_\delta(x)$ we have $\mathrm{res}(\mathrm{on},\mathrm{off})(x') \in \mathrm{res}(\mathrm{on},\mathrm{off})(x) + B_\epsilon$.
3. If both ϕ and ψ are compact, then F is continuous.
4. We have that F is separated, iff $\phi \cap [0,1] = \phi \cap \psi = \emptyset$ and $\phi \cap \{2\} = \emptyset$.

To verify continuity of the reset functions in practice, note that Scott continuity is preserved by function composition, hence all combinations of Scott continuous functions will be Scott continuous. In particular, we note that the following functions are Scott continuous, and thus can be used as building blocks for reset functions.

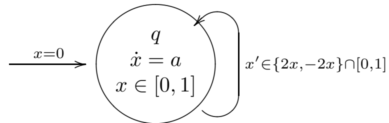
Proposition 2. *Suppose* $\mathsf{A}, \mathsf{B} \in \mathbf{U}^{\top}\mathbb{R}^n$.

1. *All step functions*
$$a \searrow b : \mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}, \quad x \mapsto \begin{cases} b & x \in a^o \\ \bot & \textit{otherwise} \end{cases}$$
are continuous for $a \in \mathbf{U}^{\top}\mathsf{A}, b \in \mathbf{U}^{\top}\mathsf{B}$*, where* a^o *denotes the interior of* a.
2. *All co-step functions*
$$a \nwarrow\!\!\searrow b : \mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}, \quad x \mapsto \begin{cases} b & x \in a \\ \top & \textit{otherwise} \end{cases}$$
are continuous for $a \in \mathbf{U}^{\top}\mathsf{A}, b \in \mathbf{U}^{\top}\mathsf{B}$.
3. *All functions*
$$\bowtie b : \mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}, \quad x \mapsto \{x\} \cap b$$
are continuous for $b \in \mathbf{U}^{\top}\mathsf{B}$
4. *If* $f_1, f_2 : \mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}$ *are continuous, then so is* $f_1 \cup f_2 : \mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}, x \mapsto f_1(x) \cup f_2(x)$.
5. *If* $(f_i)_{i\in I}$ *is directed (w.r.t. the pointwise ordering), then* $\bigsqcup_{i\in I} f_i : \mathsf{A} \to \mathbf{U}^{\top}\mathsf{B}, x \mapsto \bigsqcup_{i\in I} f_i(x)$ *is continuous.*

The previous proposition gives some general construction principles for continuous hybrid automata, and can be applied to show that a large class of flow automata are actually continuous. We now turn to the separation property. The following example, which is a variation of the bouncing ball automaton [19] shows, that the separation property is vital for the computability of the semantic function associated with a flow automaton.

Example 2. Consider the automaton $F = (Q, \mathrm{init}, \mathrm{flow}, \mathrm{res}, \mathrm{inv})$ with

- $Q = \{q\}$
- $\mathrm{inv}(q) = [0,1]$
- $\mathrm{flow}(r,t) = r + a \cdot t$
- $\mathrm{res}(x) = \{2x, -2x\} \cap [0,1]$
- $\mathrm{init}(q) = \{0\}$

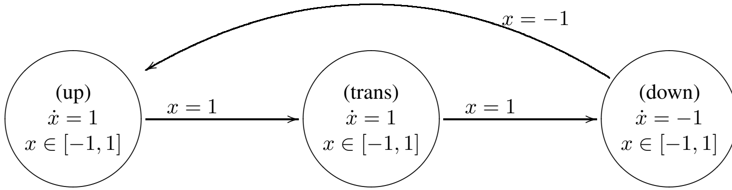


where $a \in \mathbb{R}$ is a computable real number, as depicted on the right above. Suppose we can effectively find a sequence of functions $R_k : [0,1] \to \mathbf{U}^{\top}[0,1]$ such that $\bigsqcup_{k\in\mathbb{N}} R_k = R_F$. Then clearly $R(1) = \{0\}$ iff $a = 0$, and $R(1) \cap [1/2, 1] \neq \emptyset$ iff $a > 0$. As $R(1) = \bigcap_{k\in\mathbb{N}} R_k(1)$, this implies that we can semi-decide whether $a = 0$. Together with a semi decision procedure for $a \neq 0$, we arrive at a decision procedure for $a = 0$, which is impossible, see e.g. [21].

Recall that a flow automaton is *zeno*, if it admits a trajectory $(t_i, q_i, f_i)_{i<\infty}$ with $\sup_{i<\infty} t_i < \infty$. The key consequence of separation, which makes it possible to compute the semantic function associated with a flow automaton, is that separated automata are non-zeno. This is the content of the next proposition.

Proposition 3. *Suppose F is separated and continuous. Then F is non zeno.*

Note that, while the fact that an automaton is separated is sufficient for it being non-zeno, the separation property is not necessary. Consider for example the automaton



with reset relations $\mathrm{res}(\mathrm{up}, \mathrm{trans}) = \mathrm{res}(\mathrm{trans}, \mathrm{down}) = \lambda x.\{x\} \cap \{1\}$ and $\mathrm{res}(\mathrm{down}, \mathrm{up}) = \lambda x.\{x\} \cap \{-1\}$ and initial state $(\mathrm{up}, 0)$. Then clearly F is non-zeno, but F is not separated. This suggests that the separation property can be relaxed, and one just needs to require that there is no finite loop $(q_0, x_0), (q_1, x_1), \ldots, (q_l, x_l)$ with $x_{i+1} \in \mathrm{res}_{q_i, q_{i+1}}(x_i)$ and $x_0 \in \mathrm{res}_{q_l, q_0}(x_l)$, but we refrain from doing so, as the technical complications would obscure the techniques at the heart of our analysis.

4 Denotational Semantics of Continuous and Separated Automata

We now turn to the main objective of the present paper and describe a computational method for obtaining R_F for a continuous and separated flow automaton F. Our technique will compute the function R_F as least fixpoint of a functional of type $([0, \infty) \Rightarrow \mathcal{U}) \to ([0, \infty) \Rightarrow \mathcal{U})$, where $\mathcal{U} = \prod_{q\in Q} \mathbf{U}^{\top} \mathrm{inv}(q)$. We first introduce some terminology to make the notation more readable.

Definition 5. *Suppose $F = (Q, \mathrm{inv}, \mathrm{flow}, \mathrm{res}, \mathrm{init})$ is a flow automaton. The function*

$$f_F : \mathcal{U} \times \mathbf{I}[0, \infty) \to \mathcal{U},$$
$$((x_q)_{q\in Q}, \alpha) \mapsto (\{\mathrm{flow}(q)(y_q, t) \mid y_q \in x_q, t \in \alpha, \forall s \leq t.\mathrm{flow}(q)(y_q, s) \in \mathrm{inv}(q)\})_{q\in Q}$$

is called the extended flow function*, and*

$$r_F : \mathcal{U} \to \mathcal{U}, (x_q)_{q\in Q} \mapsto (\bigcup_{p\in Q} \mathcal{E}(\mathrm{res}(p, q))(x_p))_{q\in Q}$$

is the extended reset function. If the automaton F is clear from the context, we omit the corresponding subscript.

While the extended reset function collects all the functions $\mathrm{res}(p, q)$ in a single map, the rationale behind the definition of the extended flow function is moreover that we need to cut out those portions of the flows that leave or re-enter a state invariant. Pictorially, this leaves us with the shaded region displayed in Figure 1. It is easy to see that both

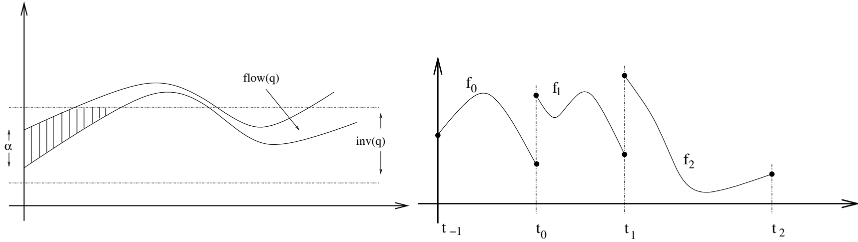


Fig. 1. The functions $f_F(\alpha, \cdot)$ (left) and $\rho^\sharp$ (right)

the extended flow function, and the extended reset function are Scott continuous.

Lemma 4. *If F is continuous, then both f_F and r_F are Scott continuous.*

With this notation, we are now ready to introduce the key concept of the present paper: the forward action associated with a flow automaton. As we will see later, the least fixpoint of this operator captures the set of of states the automaton can engage in at time t and, moreover, can be effectively computed.

Definition 6. *Suppose F is a flow automaton. The operator*

$$\Phi_F : ([0, \infty) \Rightarrow \mathcal{U}) \to ([0, \infty) \Rightarrow \mathcal{U}), \rho \mapsto \lambda t. f_F(i_F, t) \cup \bigcup_{s \le t} f_F(r_F(\rho(s), t - s))$$

is called the forward action *associated with F.*

The forward action combines the discrete action and the continuous flow, and can be seen as a generalisation of the fixpoint operator associated with an IFS [6]. Our goal is to show that the least fixpoint of the forward action is precisely the function R_F that computes reachable states. In order to compute this fixpoint effectively, we first have to ensure that Φ_F is compatible with approximations, i.e. Φ_F is well-defined and Scott continuous.

Lemma 5. *Both $\Phi_F(\rho)$, for $\rho \in ([0, \infty) \Rightarrow \mathcal{U})$, and Φ_F are Scott continuous.*

Continuity of Φ_F now guarantees the existence of a least fixpoint of Φ_F, which we denote by $[\![F]\!]$ throughout. We now examine this fixpoint and show that it precisely captures the set of all F-trajectories.

In order to show soundness, it is convenient to formulate trajectories as maps into the upper space. In order to turn the trajectories into Scott continuous functions, we let the induced function take a non-singleton set as value whenever the discrete control state changes. Below, $\mathrm{sgn}(x) \in \{-1, 0, +1\}$ is the sign of $x \in \mathbb{R}$.

Lemma 6. *Suppose $f_{-1} : [-1,0] \to \mathbb{R}$ and $f_{+1} : [0,1] \to \mathbb{R}$ are continuous. Then the function $f \oplus g : [-1,1] \to \mathbf{U}^{\top}\mathbb{R}$, defined by $t \mapsto \{f_{\mathrm{sgn}(t)}(t)\}$, if $t \neq 0$, and $t \mapsto \{f(0), g(0)\}$ otherwise, is Scott continuous.*

For F-trajectories, we have the following corollary. Note that the condition on trajectories is automatic for continuous and separated automata.

Corollary 7. *Suppose F is a flow automaton $\rho = (t_i, q_i, f_i)_{i<N}$ is a F-trajectory with $\sup_i t_i = \infty$ in case $N = \infty$. Then*

$$\rho^\sharp : [0,\infty) \to \mathcal{U}, \quad t \mapsto \{(q_i, f_i(t)) \mid t \in [t_{i-1}, t_i]\}$$

where $q \in Q$, is Scott-continuous. Moreover, $R_F(t) = \bigcup\{\rho^\sharp(t) \mid \rho \text{ is an } F-\text{trajectory}\}$, if F is a flow automaton.

The function $\rho^\sharp$ is visualised on the right hand side of Figure 1. The next statement is a stepping stone for proving the soundness of our approach. We show, that applying Φ_F, we do not lose any trajectories; hence starting the fixpoint iteration from the everywhere undefined function, the least fixpoint is guaranteed to cover all trajectories.

Lemma 8. *Suppose F is separated and continuous, ρ is an F-trajectory and $\sigma \in ([0,\infty) \Rightarrow \mathcal{U})$ with $\sigma \sqsubseteq \rho^\sharp$. Then $\Phi_F(\sigma) \sqsubseteq \rho^\sharp$.*

Note that the proof of the previous theorem relies on separatedness. Using the above result, correctness of the fixpoint construction is very easy to show.

Corollary 9 (Correctness). *Suppose F is continuous and separated. Then $s \in [\![F]\!](t)$ if $\mathrm{init} \to_*^t s$ for all $s \in S_F$ and all $t \in [0,\infty)$.*

While the previous result asserts soundness, we now turn to computational adequacy of the construction, i.e. we show that $R_F = [\![F]\!]$, where $[\![F]\!]$ is the least fixpoint of Φ_F.

Theorem 10 (Computational Adequacy). *Suppose F is separated and continuous. Then $s \in [\![F]\!](t)$ iff $\mathrm{init} \to_*^t s$ for all $s \in S_F$ and all $t \geq 0$.*

The proof of the theorem in fact demonstrates, that any function $\rho \in ([0,\infty) \Rightarrow \mathcal{U})$ with $\rho \sqsubseteq [\![F]\!]$, that does not arise as an F-trajectory, necessarily leads to a violation of the separatedness property. Unfolding the definition of V_F, we also obtain computational means to obtain the states of a flow automaton F that can be visited up to time t, in terms of the least fixpoint $[\![F]\!]$ of the forward action associated with F. This then gives $V_F(t) = \bigcup_{s \leq t} R_F(s)$. However, we can also obtain V_F as a fixpoint of an operator in its own right.

Definition 7. *The operator*

$$\Psi_F : ([0,\infty) \Rightarrow \mathcal{U}) \to ([0,\infty) \Rightarrow \mathcal{U}), \rho \mapsto f_F(i_F, [0,t]) \cup \bigcup_{s \leq t} f_F(r_F(\rho(s), [0, t-s]))$$

is the visited states operator *associated with F.*

The properties of Ψ_F are similar to those of Φ_F, in particular, Ψ_F is Scott continuous, and the least fixpoint captures the set of visited states. More formally:

Theorem 11. *Suppose $\rho : [0,\infty) \to \mathcal{U}$ is the least fixpoint of Ψ_F. Then $\rho = V_F$.*

While Theorem 10 and Theorem 11 are important on their own, as they allow us to obtain the semantics of hybrid automata as a least fixpoint in a suitable function space, they also allow us to derive new results about the function R_F that yields the states reachable at time t for continuous and separated automata:

Corollary 12. *1. $R_F(t)$ and $V_F(t)$ are compact for every $t \in [0,\infty)$.*
2. R_F and V_F are Scott continuous.

5 Approximation of Flow Automata

We have seen that the semantics $[\![F]\!] : [0,\infty) \to \mathcal{U}$ of a flow automaton F can be computed as the least fixpoint of a functional on $([0,\infty) \Rightarrow \mathcal{U})$. While this gives a mathematical means of understanding the semantics, we now show, that this also induces a method to compute the semantics up to an arbitrary degree of accuracy.

To do this, we restrict attention to countable bases of the involved domains, that is, to finitely representable objects, that generate all of the involved domains by means of directed suprema. We show, that we can effectively compute the least fixpoint of the functional up to an arbitrary degree of accuracy, if we approximate all continuous ingredients of the automaton. We begin by introducing the bases of the domains we are interested in. For the remainder of the section, we fix a countable dense ordered subring $D = \{d_0, d_1, \dots\}$ with decidable equality and order, and computable ring operations. We put $D_k = \{d_0, \dots, d_k\}$. We only treat the case of computing R_F as a least fixpoint; the setup can be easily adapted to accommodate also V_F.

Definition 8. *We let, for an arbitrary set $S \subseteq \mathbb{R}$, $\mathbf{I}\mathbb{R}^n_S = \{[a_1,b_1] \times \cdots \times [a_n,b_n] \in \mathbf{I}\mathbb{R}^n \mid a_1,\dots,a_n,b_1,\dots,b_n \in S\} \cup \{\mathbb{R}\}$ denote the set of rectangles with endpoints in S, augmented with the least element $\mathbb{R}$. If $\mathsf{A} \subseteq \mathbb{R}^n$ is a semi rectangle, then $\mathbf{I}\mathsf{A}_S = \{\mathsf{A} \cap b \mid b \in \mathbf{I}\mathbb{R}^n_S\}$ denotes the set of rectangles $R \in \mathbf{I}\mathbb{R}^n$ that are contained in A and have corners in S, again with a bottom element. We distinguish two kinds of step functions:*

$$a \searrow^i b : \mathsf{A} \to \mathsf{B}, x \mapsto \begin{cases} b & x \in a^o \\ \bot & \text{otherwise,} \end{cases} \quad \text{and} \quad a \searrow b : \mathsf{A} \to \mathsf{B}, x \mapsto \begin{cases} b & a \ll x \\ \bot & \text{otherwise} \end{cases}$$

where B is a dcpo with $b \in \mathsf{B}$ in both cases; $\mathsf{A} \subseteq \mathbb{R}^n$ is a semi rectangle with $a \in \mathbf{I}\mathsf{A}$ in the case of $a \searrow^i b$, and A is a dcpo with $a \in \mathsf{A}$ for $a \searrow b$. We use the following bases:

1. *If $\mathsf{A} \subseteq \mathbb{R}^n$ is a semi-rectangle with corners in $D \cup \{\pm\infty\}$, then the set $\mathbf{I}\mathsf{A}_D$ of rectangles contained in A and corners in D is called the* standard base *of $\mathbf{I}\mathbb{R}^n$.*
2. *If $\mathsf{A} \in \mathbf{I}\mathbb{R}^n_D$, then the set $\mathbf{U}^\top \mathsf{A}_D = \{\cup_{1 \le i \le k} D_i \mid i \in \mathbb{N}, D_i \in \mathbf{I}\mathsf{A}_D\}$ of finite unions of rectangles with corners in D is the* rectangular base *of $\mathbf{U}^\top \mathsf{A}$.*
3. *If $(A_i)_D$ is a base of the dcpo A_i, then $(A_1 \times \cdots \times A_n)_D = (A_1)_D \times \cdots \times (A_k)_D$ is the base of $A_1 \times \cdots \times A_k$ induced by the components.*
4. *If A_D and B_D are bases of the dcpos A and B, respectively, then $(A \Rightarrow \mathsf{B})_D = \{\bigsqcup_{i \le i \le k} a_i \searrow b_i \in (\mathsf{A} \Rightarrow \mathsf{B}) \mid a_1,\dots,a_k \in \mathsf{A}_D, b_1,\dots b_k \in \mathsf{B}_D\}$ is the rectangular base of $(\mathsf{A} \Rightarrow \mathsf{B})$.*

5. *Finally, if* $\mathsf{A} \subseteq \mathbb{R}^n$ *is a semi rectangle with corners in* $D \cup \{\pm\infty\}$ *and* B_D *is a base of the dcpo* B*, then* $(\mathsf{A} \Rightarrow \mathsf{B})_D = \{\bigsqcup_{1 \le i \le k} a_i \searrow^i b_i \in (\mathsf{A} \Rightarrow \mathsf{B}) \mid a_1, \ldots, a_k \in \mathbf{I}\mathsf{A}_D, b_1, \ldots, b_k \in \mathsf{B}_D$ *is the induced base of of* $(\mathsf{A} \Rightarrow \mathsf{B})$

where we indicate by $\bigsqcup_{1 \le i \le k} a_i \searrow b_i \in (\mathsf{A} \Rightarrow \mathsf{B})$ *that we consider only consider consistent step functions [8, Section 2], similarly for* $\bigsqcup_{1 \le i \le k} a_i \searrow^i b_i$.

In words, if $\mathsf{A}, \mathsf{B} \subseteq \mathbb{R}^n$ are semi-rectangles, $\mathbf{I}\mathsf{A}_D$ is the set of rectangles contained in A with corners in D and $\mathbf{U}^\top \mathsf{A}_D$ is the set of finite unions of rectangles with corners in D. For the function space, $(\mathsf{A} \Rightarrow \mathbf{U}^\top \mathsf{B})_D$ is the induced base of the space of functions of one or more real variables; $(\mathbf{U}^\top \mathsf{A} \Rightarrow \mathbf{U}^\top \mathsf{B})_D$ is the induced base of the function space of a compact set valued variable.

It is easy to see that the sets introduced above are indeed bases of the corresponding domain. We now use these bases to show, that the fixpoint operator Φ_F associated to a flow automaton can be effectively computed, given approximations of the components of the automaton. In order to make assertions about the computability of functions in the domain theoretic model of computation, we have to fix an enumeration of the base of the involved domains. We do not do this explicitely here, but instead assume that the bases $(\cdot)_D$ above come with an effective enumeration $\iota : \mathbb{N} \to (\cdot)_D$, which is fix throughout. In particular, the enumeration gives rise to a notion of *effective sequence*: If A is a dcpo whose base is enumerated via $\iota : \mathbb{N} \to \mathsf{A}_D$, then a sequence $(a_k)_{k \in \mathbb{N}}$ in A_D is *effective*, if $a_k = \iota(f(k))$ for some total recursive function f.

First, note that composition of base functions yields a base function, and that the extension function is effectively computable.

Lemma 13. *Suppose* $f \in (\mathsf{A} \Rightarrow \mathbf{U}^\top \mathsf{B})_D$ *and* $g \in (\mathbf{U}^\top \mathsf{B} \Rightarrow \mathbf{U}^\top \mathsf{C})_D$*. Then*

1. $g \circ f \in (\mathsf{A} \Rightarrow \mathbf{U}^\top \mathsf{C})_D$ *and* $g \circ f$ *is effectively computable.*
2. $\mathcal{E}(f) \in (\mathbf{U}^\top \mathsf{A} \Rightarrow \mathbf{U}^\top \mathsf{B})_D$ *and* $\mathcal{E}(f)$ *is effectively computable.*

The next lemma gives a basis representation of subtraction, which is needed in the definition of the fixpoint functional Φ_F associated with F, see Definition 6.

Lemma 14. *The functions* $M_k : [0, \infty)^2 \to \mathbf{I}[0, \infty)$*, defined by* $M_k = \bigsqcup\{a \times b \searrow b - a \mid a, b \in \mathbf{I}[0, \infty)_{D_k}\}$ *satisfy* $M_k \in ([0, \infty)^2 \Rightarrow \mathbf{I}[0, \infty))_D$ *for all* $k \in \mathbb{N}$*, and* $\bigsqcup_{k \in \mathbb{N}} M_k = \lambda(x, y).y - x$.

Building on these basic facts, we can now show, that the least fixpoint of the operator Φ_F associated with a flow automaton is effectively computable. This of course hinges on the fact that the automaton is effectively given:

Definition 9. *Suppose* $F = (Q, \mathrm{init}, \mathrm{flow}, \mathrm{res}, \mathrm{inv})$ *is a flow automaton. We say that* F *is* effectively given *if it comes with*

- *an effective sequence* $(i_k^q)_{k \in \mathbb{N}}$ *in* $(\mathbf{U}^\top \mathrm{inv}(q))_D$ *with* $\bigsqcup_{k \in \mathbb{N}} i_k^q = \mathrm{init}(q)$
- *an effective sequence* $(f_k^q)_{k \in \mathbb{N}}$ *in* $(\mathbb{R}^n \times [0, \infty) \Rightarrow \mathbf{I}\mathbb{R}^n)_D$ *with* $\bigsqcup_{k \in \mathbb{N}} f_k^q = \mathrm{flow}(q)$
- *an effective sequence* $(r_k^{p,q})_{k \in \mathbb{N}}$ *in* $(\mathrm{inv}(p) \Rightarrow \mathbf{U}^\top \mathrm{inv}(q))_D$ *with* $\bigsqcup_k r_k^{p,q} = \mathrm{res}(p, q)$

for all $q \in Q$*, resp. all* $(p, q) \in Q^2$ *and* $\mathrm{inv}(q) \in \mathbf{U}^\top \mathbb{R}^n_D$ *for all* $q \in Q$*. The family of sequences* $(f^q_k)_{k\in\mathbb{N}}$*,* $(i^q_k)_{k\in\mathbb{N}}$ *(where* $q \in Q$*) and* $(r^{p,q}_k)_{k\in\mathbb{N}}$ *(where* $(p, q) \in Q^2$*) are called an* effective presentation *of* F*.*

That is to say that, for an effectively given flow automaton, the initial states, the flow functions and the reset functions are computable. It is easy to see that every effectively given flow automaton induces a computable extended flow function, and a computable extended reset function. Spelling this out, Lemma 13 and Lemma 14, together with the Scott continuity of Φ_F, allow us to prove our second main theorem:

Theorem 15. *Suppose* F *is an effectively given flow automaton. Then we can effectively obtain a sequence* (σ_k) *with* $\bigsqcup_{k\in\mathbb{N}} \sigma_k = [\![F]\!]$*.*

6 Hybrid Automata

In this section, we transfer our results on flow automata to hybrid systems, where the continuous behaviour of the system in every given control state is described directly by a vector field. This is achieved by associating the equivalent flow automaton to the hybrid automaton under consideration. If the hybrid automaton is effectively given, we show, that the same also holds for the induced flow automaton. We thus obtain an effective framework for the analysis of hybrid automata. The following is a variant of the standard definition of a hybrid automaton [12, 2].

Definition 10. *A* hybrid automaton *is a tuple* $H = (Q, \mathrm{inv}, \mathrm{vect}, \mathrm{res}, \mathrm{init})$ *where* $Q, \mathrm{inv}, \mathrm{res}, \mathrm{init}$ *are as in Definition 2, and* $\mathrm{vect} = (\mathrm{vect}_q)_{q\in Q}$ *is a family of vector fields* $\mathrm{vect}(q) : \mathbb{R}^n \to \mathbb{R}^n$ *where all* $\mathrm{vect}(q)$ *are assumed to be globally Lipschitz, i.e.* $\|\mathrm{vect}(q)(x) - \mathrm{vect}(q)(y)\| \leq L\|x - y\|$*, for all* $q \in Q$*,* $x, y \in \mathbb{R}^n$ *and some* $L \in \mathbb{R}$*.*

In contrast to the standard definition, the trajectories of the real variables are described by a differential equation rather than differential inclusion. We require this restriction in view of the domain theoretic treatment of differential equations [9], which in general gives a strict over-approximation to the solution of differential inclusion.

We recall from Lemma 1, that every Lipschitz vector field $v : \mathbb{R}^n \to \mathbb{R}^n$ induces a flow function $f : \mathbb{R}^n \times [0, \infty) \to \mathbb{R}^n$. The main reason for restricting attention to vector fields that are globally Lipschitz is that the induced flows are globally defined; we believe that similar results can be obtained for vector fields whose associated flows don't diverge. Replacing the vector field by the induced flow function, every hybrid automaton H induces a flow automaton F; in this case, we write $[\![H]\!]$ for $[\![F]\!]$.

Definition 11. *Suppose* $H = (Q, \mathrm{inv}, \mathrm{vect}, \mathrm{res}, \mathrm{init})$ *is a hybrid automaton and* $\mathrm{flow}(q)$ *is the flow induced by* $\mathrm{vect}(q)$*. The automaton* $F = (Q, \mathrm{inv}, \mathrm{flow}, \mathrm{res}, \mathrm{init})$ *is called the* flow automaton induced by H*. We say that* H *is continuous (resp. separated), if the induced flow automaton is continuous (resp. separated). We say that* H *is* effectively given *if it comes with*

- *an effective sequence* $(i^q_k)_{k\in\mathbb{N}}$ *in* $(\mathbf{U}^\top \mathrm{inv}(q))_D$ *with* $\bigsqcup_{k\in\mathbb{N}} i^q_k = \mathrm{init}(q)$
- *an effective sequence* $(v^q_k)_{k\in\mathbb{N}}$ *in* $(\mathbf{I}\mathbb{R}^n \Rightarrow \mathbf{I}\mathbb{R}^n)_D$ *with* $\bigsqcup_{k\in\mathbb{N}} v^q_k = \mathrm{vect}(q)$

– *an effective sequence* $(r_k^{p,q})_{k\in\mathbb{N}}$ *in* $(\mathrm{inv}(p) \Rightarrow \mathbf{U}^{\top}\mathrm{inv}(q))_D$ *with* $\bigsqcup_k r_k^{p,q} = \mathrm{res}(p,q)$

for all $q \in Q$*, resp. all* $(p,q) \in Q^2$ *and* $\mathrm{inv}(q) \in \mathbf{U}^{\top}\mathbb{R}^n_D$ *for all* $q \in Q$*. The family of sequences* $(i_k^q)_{k\in\mathbb{N}}$*,* $(v_k^q)_{k\in\mathbb{N}}$ *(where* $q \in Q$*) and* $(r_k^{p,q})_{k\in\mathbb{N}}$ *(where* $(p,q) \in Q^2$*) are called an* effective presentation *of* H*.*

We have seen in Theorem 15, that the function $[\![F]\!]$ associated with a flow automaton, which captures the states reachable by F at time $t \in [0,\infty)$, is effectively computable, if F is effectively given. In order to associate an effectively given flow automaton to an effectively given hybrid automaton, we therefore have to produce approximations $f_k \in (\prod_{q\in Q}\mathrm{inv}(q)\times[0,\infty) \Rightarrow \mathcal{U})$ of the flow function induced by a hybrid automaton. In other words, we have to solve the initial value problems defined by the vector field that defines the hybrid automation. This is achieved by instantiating results from [9, 10], where it is shown how to solve initial value problems in a domain theoretic framework.

Theorem 16. *Suppose* H *is an effectively given hybrid automaton. Then so is the induced flow automaton* F*. Moreover, we can construct an effective presentation of* F *from an effective presentation of* H*.*

Together with Theorem 15, we have now shown, that the semantic function $[\![H]\!]$, associated with an effectively given hybrid automaton, is computable:

Theorem 17. *Suppose* H *is effectively given, continuous and separated. Then the function* $[\![H]\!] : [0,\infty) \to \mathcal{U}$ *is effectively computable.*

Moreover, as all our constructions are based on bases of the domains involved, the algorithms underlying Theorems 17 and 15 are based on proper data types, and can be directly implemented on a digital computer: we choose the dyadic (or rational) numbers for D, and then define data types that directly represent the bases $[0,\infty)_D$ and $\mathcal{U}_D$, as well as the bases of the function space $([0,\infty) \Rightarrow \mathcal{U})_D$. Computing with dyadic (or rational) numbers then allows us to manipulate elements of the data types without any loss of arithmetical precision. Moreover, we have shown that the fixpoint operator, that gives rise to the semantic function $[\![H]\!]$ of a hybrid automaton, can be effectively computed on the described data types.

Conclusions and Future Work. Of course, much remains to be done. While the presentation in this paper is geared towards demonstrating that a domain theory has all the necessary tools to facilitate the algorithmic analysis of hybrid automata, we anticipate that major improvements will be made on the efficiency of the involved algorithms. In particular, we are working towards rigorous estimates of the convergence speed and the complexity of the described fixpoint algorithms in terms of the Hausdorff distance in $\mathcal{U}$, which will also allow us to make concrete assertions about the computational complexity of our method. For now, we have concentrated on computing the semantic function $[\![H]\!]$ associated with a hybrid automaton. Future work will bring a framework for computing the set of reachable states of a hybrid automaton, and a real time logic with associated model checking procedure for the automated verification of hybrid automata.

References

1. S. Abramsky and A. Jung. Domain Theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3. Clarendon Press, 1994.
2. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoret. Comp. Sci*, 138(1):3–34, 1995.
3. R. Alur, T. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.
4. J.-P. Aubin. *Viability Theory*. Birkhäuser, 1991.
5. A. Edalat. Dynamical systems, measures and fractals via domain theory. *Information and Computation*, 120(1):32–48, 1995.
6. A. Edalat. Power domains and iterated function systems. *Information and Computation*, 124:182–197, 1996.
7. A. Edalat, M Krznarić, and A. Lieutier. Domain-theoretic solution of differential equations (scalar fields). In *Proceedings of MFPS XIX*, volume 83 of *Elect. Notes in Theoret. Comput. Sci.*, 2004.
8. A. Edalat and A. Lieutier. Domain theory and differential calculus (functions of one variable. *Math. Struct. Comp. Sci.*, 14, 2004.
9. A. Edalat and D. Pattinson. A domain theoretic account of picard's theorem. In *Proc. ICALP 2004*, number 3142 in Lect. Notes in Comp. Sci., pages 494–505, 2004.
10. A. Edalat and D. Pattinson. Domain theoretic solutions of initial value problems for unbounded vector fields. In M. Escardó, editor, *Proc. MFPS XXI*, Electr. Notes in Theoret. Comp. Sci., 2005. to appear.
11. G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. Scott. *Continuous Lattices and Domains*. Cambridge University Press, 2003.
12. T. Henzinger. The theory of hybrid automata. In M. Inan and R. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series F: Computer and Systems Sciences*, pages 265–292. Springer Verlag, 2000.
13. T. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.
14. T. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.
15. T. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HYTECH: Hybrid systems analysis using interval numerical methods. In *Proc. HSCC 2000*, volume 1790 of *Lect. Notes in Comp. Sci.*, pages 130–144. Springer, 2000.
16. J. E. Hutchinson. Fractals and self-similarity. *Indiana University Mathematics Journal*, 30:713–747, 1981.
17. J. Lygeros, D. Godbole, and S. Sastry. Verified hybrid controllers for automated vehicles. *IEEE Transactions on Automatic Control*, 43(4):522–539, 1998.
18. O. Müller and T. Stauner. Modelling and verification using linear hybrid automata – a case study. *Mathematical and Computer Modelling of Dynamical Systems*, 6(1):71–89, 2000.
19. S. Simic, K. Johansson, S. Sastry, and J. Lygeros. Towards a geometric theory of hybrid systems. In N. Lynch and B. Krogh, editors, *Proc. HSCC 2000*, volume 1790 of *Lect. Notes in Comp. Sci.*, pages 421–436, 2000.
20. C. Tomlin, G. Pappas, and S. Sastry. Conflict resolution for air traffic management : A study in muti-agent hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):509–521, 1998.
21. K. Weihrauch. *Computable Analysis*. Springer, 2000.

Reversing Algebraic Process Calculi

Iain Phillips[1] and Irek Ulidowski[2]

[1] Department of Computing, Imperial College London, England
iccp@doc.ic.ac.uk
[2] Department of Computer Science, University of Leicester, England
iu3@mcs.le.ac.uk

Abstract. Reversible computation has a growing number of promising application areas such as the modelling of biochemical systems, program debugging and testing, and even programming languages for quantum computing. We formulate a procedure for converting operators of standard algebraic process calculi such as CCS, ACP and CSP into reversible operators, while preserving their operational semantics.

1 Introduction

Reversible computation has a growing number of promising application areas such as the modelling of biochemical systems [8], program debugging and testing [20], and even programming languages for quantum computing [2]. Landauer [15] showed how irreversible computation generates heat; the efficient operation of future miniaturised computing devices could depend on exploiting reversibility [7]. We have been inspired to look at this area by the work of Danos and Krivine on reversible CCS [8, 9, 10] and Abramsky on mapping functional programs into reversible automata [1].

We wish to investigate reversibility for algebraic process calculi in the style of CCS [16], with Structural Operational Semantics (SOS) [19] rules. Given a forward *labelled transition relation* (ltr) $\rightarrow$ we are interested in obtaining a *reverse ltr* $\rightsquigarrow$ which is the inverse of $\rightarrow$. This can always be done, but if we just reverse a standard process language we end up with too many possibilities, since processes do not "remember" their past states. Danos and Krivine solve this problem by storing "memories" of past behaviour which are carried along with processes. Memories also keep track of which thread or threads performed an action. This has the effect that backtracking does not have to follow the exact order of forward computation in reverse. To take a simple example, suppose that the process $a.b \mid c$ performs a followed by b and then c (here "." and "|" are the prefixing and the parallel composition of CCS, respectively). The process can backtrack by reversing b, then a and finally c. However, a cannot be reversed before b has been reversed.

We wish to produce reversible process calculi without relying on external devices such as memories. Our starting point is that irreversibility in a language such as CCS comes from the consumption of guards and alternative choices.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 246–260, 2006.
© Springer-Verlag Berlin Heidelberg 2006

We therefore decide to leave these in place, so that process structure remains fixed throughout a computation. Returning to the example of $a.b \mid c$, we might let the state after a, b and c have been performed be denoted by $\underline{a}.\underline{b} \mid \underline{c}$, where the underlined actions are *past* actions. There is plainly just the right amount of information here to reverse the process, while allowing $\underline{c}$ to be reversed independently of $\underline{b}$ and $\underline{a}$. This approach allows us also to keep track of unused alternatives discarded during computation. Consider $a.b + c$, where "+" is the choice operator of CCS. After the initial a, the alternative c is discarded and we can only proceed with b. This state is represented as $\underline{a}.b + c$; it is clear which alternative was taken and what will happen next.

Reversibility can help to some extent to distinguish concurrency from causation. In the reversible world, Milner's expansion law does not hold: we have $a \mid b \neq a.b + b.a$ since $a \mid b \xrightarrow{a}\xrightarrow{b} \underline{a} \mid \underline{b} \stackrel{a}{\rightsquigarrow}$ but $a.b + b.a \xrightarrow{a}\xrightarrow{b} \underline{a}.\underline{b} + b.a \stackrel{a}{\not\rightsquigarrow}$.

When we come to consider autoconcurrency and communication we find that the simple method just outlined arguably discards too much information. For instance, the processes $a \mid a$ and $a.a$ cannot be distinguished by the above argument, as we are not able to tell apart the two occurrences of a. Moreover, the process $a \mid \overline{a}$ can evolve by a communication between a and the complementary action $\overline{a}$ to yield $\underline{a} \mid \underline{\overline{a}}$. This state could also have been reached by performing the two actions separately, and there is nothing in the notation to stop us from undoing the two actions separately. But if the communication represents a binding between two (biological) entities, then such separate backtracking of a and $\overline{a}$ is not reasonable.

Our solution is to use a more expressive form of past actions, where each occurrence of action a is "marked" by a fresh identifier m and written as $a[m]$. Also, we insist that the two parties to a communication between action a and $\overline{a}$ agree on this identifier or *communication key* which is unique to that communication. This means that a and $\overline{a}$ are now locked together and can only be undone together. Now, we can deal with the autoconcurrency example. Process $a \mid a$ can perform the a actions with keys m and n to produce $a[m] \mid a[n]$, and then reverse these actions in any order. However, $a.a$ cannot match this behaviour: after the a actions with keys m and n, the process $a[m].a[n]$ cannot reverse on $a[m]$.

We propose a method for reversing process operators that are definable by SOS rules in a general format. As far as we are aware, this is the first time this has been done for algebraic process calculi. As we have described informally above, we rely on reformulating operators of standard process calculi into new operators that can be easily reversed, while preserving their operational meaning. In this paper we attempt to balance the generality of the format on one hand and the technical simplicity of the proposed method on the other hand. The chosen format is general enough for the definitions of the majority of useful process operators, and the method presented is intuitive and easy to apply.

Our format is a subformat of the *path* format [3] and consists of *dynamic* rules, where the operator is destroyed by a transition, and *static* rules, where the operator remains present after the transition. Reversing static rules is easier because they preserve the context during execution. Dynamic rules, however, consume

the context, removing the unused alternatives. The kernel of our method is to transform dynamic rules into static-like rules. *Auxiliary operators* and *predicates* are used to keep the structure of terms unchanged and to enforce correct use of subterms in the reformulated contexts. Once SOS rules for operators are reformulated as above, the reverse SOS rules are obtained simply as symmetric versions of the forward rules.

As an illustration of the method we consider the CCS choice operator +. We reformulate it as a static operator and use predicate std, meaning that the argument is a standard term that uses no past actions (and no keys), to control when arguments can fire in rules. The reverse rules (on the right) for the converted + are then obtained by symmetry:

$$\frac{X \xrightarrow{a} X' \quad \mathsf{std}(Y)}{X + Y \xrightarrow{a} X' + Y} \quad \frac{Y \xrightarrow{a} Y' \quad \mathsf{std}(X)}{X + Y \xrightarrow{a} X + Y'} \qquad \frac{X \overset{a}{\rightsquigarrow} X' \quad \mathsf{std}(Y)}{X + Y \overset{a}{\rightsquigarrow} X' + Y} \quad \frac{Y \overset{a}{\rightsquigarrow} Y' \quad \mathsf{std}(X)}{X + Y \overset{a}{\rightsquigarrow} X + Y'}$$

We prove a number of results to show that our method yields well-behaved transition relations. We show that the new forward ltr is conservative over the standard ltr (Theorem 5.8). Also the new forward and reverse ltrs satisfy certain confluence properties (Propositions 5.4 and 5.5). The processes which are reachable from standard processes by forward-only transitions are closed under reverse transitions, meaning that a process can never reverse into an "inconsistent" past (Proposition 5.6). We also formulate a notion of *forward-reverse bisimulation*, which is a congruence (Theorem 6.7).

The rest of the paper is structured as follows. In Section 2 we define the simple process calculi which we shall be making reversible, and in Section 3 we describe our procedure for generating the new reversible calculi. In Section 4 we illustrate our method by applying it to CCS. We also discuss related work, and in particular RCCS [9]. In Section 5 we prove various results about the new reversible transition relations, and in Section 6 we define an appropriate notion of bisimulation. Section 7 indicates how to adapt the method to a more general format that contains constants and predicates. We end with some conclusions.

The proofs of the presented results, further results, examples and discussion are available in the full version of this paper [18].

2 Process Calculi

In this section we describe the process calculi to which we shall apply our procedure for generating reversible calculi.

A *signature* is a set Σ of operator symbols, each with a particular arity. The set of *terms* over Σ is denoted by $T(\Sigma)$. We shall tend to refer to terms as *processes*. We let $P, Q, \ldots$ range over processes.

A *process calculus* $L = (\Sigma, A, R)$, is given by a signature Σ, a set of actions A and a set R of SOS rules. We shall apply our procedure to a "standard" calculus $L_{\mathrm{S}} = (\Sigma_{\mathrm{S}}, \mathsf{Act}, R_{\mathrm{S}})$. Its terms are called *standard* terms and are denoted by Std. We shall assume that the only operator of arity zero (i.e. constant) is the deadlocked process $\mathbf{0}$. We let $f, \ldots$ range over Σ_{S}; $a, b, c, \ldots$ range over Act.

We next describe the rules R and their operational semantics.

The SOS theory gives us the flexibility and the benefits of working with whole classes of process calculi rather than with individual process calculi that are limited to a small number of operators. Typically, a class of operators is defined by a format of SOS rules that can be used to define them operationally. In this paper we shall consider simple *path* rules without copying [3]. More specifically, our rules will be mostly of the simpler *pxyft* and *pxyf* forms, where terms in the premises are variables and the source of the conclusion is a term constructed with a single operator.

Definition 2.1. Simple path *(forward) rules are expressions of the form*

$$\frac{\{\ X_i \xrightarrow{a_i} X'_i\ \}_{i\in I} \quad \{\ \mathsf{p_j}(X_j)\ \}_{j\in J}}{f(X_1,\ldots,X_n) \xrightarrow{a} t(X'_1,\ldots,X'_n)} \qquad \textit{and} \qquad \frac{\{\ \mathsf{p_j}(X_j)\ \}_{j\in J}}{\mathsf{p}(f(X_1,\ldots,X_n))}$$

where all variables X_i (X_j) and X'_i are distinct, and variables X'_i are such that $X'_i = X_i$ when $i \notin I$. Moreover, $I, J \subseteq \{1,\ldots,n\}$.

The sets of transitions and predicate expressions above the horizontal bars in the rules above are called premises*. Let r be the first rule above. Operator f is the* operator *of r. The transition below the bar in r is the* conclusion *of r. Action a in the conclusion is the* action *of r and $f(X_1,\ldots,X_n)$ and $t(X'_1,\ldots,X'_n)$ are the* source *and* target *of r, respectively. The i-th argument is* active *in r if r has a transition for X_i in the premises. The i-th argument of f is* active *if it is active in some rule for f. In the second rule,* p *is the* predicate *of the rule and the predicate expression below the bar is the* conclusion*.*

With any calculus $L = (\Sigma, A, R)$, all of whose rules are in simple *path* format, we associate an ltr $\rightarrow$ with labels A, together with a set of predicates, in the standard way; for details see [3]. Our standard calculus L_{S} will have all its rules R_{S} in simple *path* format. It will have no predicates in its rules. We shall write its ltr as $\rightarrow_{\mathrm{S}}$, and use this in writing down its rules for clarity.

We now define the precise form of SOS rules that operators of L_{S} can have. Consider an n-ary operator $f \in \Sigma_{\mathrm{S}}$ ($n \geq 1$). The set of arguments of f is $N_f = \{1,\ldots,n\}$. Operator f can have three kinds of rules: static rules, choice rules and choice axioms. We describe each in turn.

Definition 2.2. Static rules *of f are of the following form, where $I \neq \emptyset$:*

$$(I) \quad \frac{\{X_i \xrightarrow{a_i}_{\mathrm{S}} X'_i\}_{i\in I}}{f(\overrightarrow{X}) \xrightarrow{a}_{\mathrm{S}} f(\overrightarrow{X'})}$$

We require that if two static rules for f have the same premises then they have the same conclusion (i.e. the action of the conclusion is unique). Let $S_f \subseteq N_f$ be the set of all arguments occurring in the premises of static rules of f, and let $E_f = N_f \setminus S_f$. Arguments in S_f are called static arguments*.*

The arguments of the CCS and CSP [14] parallel composition operators are static, as are those of the CCS restriction and relabelling operators and the CSP hiding operator.

Next we describe the choice rules.

Definition 2.3. *A* choice rule *of f is a rule of the following form:*

$$\text{(II)} \quad \frac{X_d \xrightarrow{a}_{\mathrm{S}} X'_d}{f(\overrightarrow{X}) \xrightarrow{a}_{\mathrm{S}} X'_d}$$

We require that $d \in E_f$. Let D_f be the set of all arguments d occurring in the premises of choice rules of f. Arguments in D_f are called dynamic *arguments. Each dynamic argument d is required to be* permissive*, meaning that for each $a \in \mathsf{Act}$ there is a rule of type (II).*

Note that $D_f \subseteq E_f$, so that a dynamic argument cannot be static.

The choice operator of CCS has two dynamic arguments, both of which are permissive. The external choice operator of CSP also has two dynamic arguments, but they are not permissive: although they have choice rules for all $a \in \mathsf{Act} \setminus \{\tau\}$, they have no such rules for the τ—the rules for τ are static (see Section 7).

We also wish to encompass operators that have choice rules with empty premises such as, for example, CCS prefixing and CSP internal choice. This leads us to the third and final type of rule:

Definition 2.4. *A* choice axiom *of f is a rule r of the following form:*

$$\text{(III)} \quad r \, \frac{}{f(\overrightarrow{X}) \xrightarrow{\mathsf{act}(r)}_{\mathrm{S}} X_{\mathsf{ta}(r)}}$$

Here $\mathsf{ta}(r)$ *is the* target argument*. We require* $\mathsf{ta}(r) \in E_f$*.*

Next, we define the class of simple process calculi that we shall reverse.

Definition 2.5. *A process operator f is* simple *if either f is the deadlocked process* $\mathbf{0}$*, or f has a nonzero arity and all its rules are as in Definitions 2.2, 2.3 and 2.4. A process calculus is* simple *if all its operators are simple.*

In what follows we omit the subscripts of the three sets of arguments where no confusion can arise.

We shall require that L_{S} is simple. Note that we leave out rules with predicates at this stage. This allows us to keep the presentation side of the work manageable. As a result, the main application of this work is to reformulate and reverse Milner's CCS, and many other operators from the process calculi ACP [4] and CSP [14] and their descendants.

3 The Procedure for Generating a Reversible Calculus

We shall transform L_{S} into an operationally equivalent calculus which is easily reversible. For this we shall need to augment the processes and reformulate the rules of L_{S}.

Let $\mathcal{K}$ be an infinite set of *communication keys* (or just *keys* for short), ranged over by $m, n, \ldots$. The set of *past actions*, or actions marked with keys, is denoted

by $\mathsf{ActK} = \mathsf{Act} \times \mathcal{K}$. We write the ordered pair (a, m) as $a[m]$. We let $\mu, \ldots$ range over ActK, and $s, t, \ldots$ range over ActK^*.

We introduce the signature Σ_{A} of auxiliary operators $f_r[m]$, where r is a rule of type (III) for an operator f of R_{S}, and $m \in \mathcal{K}$. We let $\Sigma_{\mathrm{SA}} = \Sigma_{\mathrm{S}} \cup \Sigma_{\mathrm{A}}$, and let $\mathsf{Proc} = T(\Sigma_{\mathrm{SA}})$. Clearly, $\mathsf{Std} \subseteq \mathsf{Proc}$.

Our reformulation and reversing method relies on auxiliary unary predicates on Proc, namely $\mathsf{std}(P)$ and $\mathsf{fsh}[m](P)$ (all $m \in \mathcal{K}$). Informally, $\mathsf{std}(P)$ holds if $P \in \mathsf{Std}$ and $\mathsf{fsh}[m](P)$ holds if key m is fresh (i.e. not used) in P. The predicates are defined below, where the last four rules are rule schemas for all relevant operators and keys, and $m \neq n$ in the last rule schema.

$$\frac{}{\mathsf{std}(\mathbf{0})} \qquad \frac{\{\mathsf{std}(X_i)\}_{i \in N}}{\mathsf{std}(f(\overrightarrow{X}))} \qquad \frac{}{\mathsf{fsh}[m](\mathbf{0})} \qquad \frac{\{\mathsf{fsh}[m](X_i)\}_{i \in N}}{\mathsf{fsh}[m](f(\overrightarrow{X}))} \qquad \frac{\{\mathsf{fsh}[m](X_i)\}_{i \in N}}{\mathsf{fsh}[m](f_r[n](\overrightarrow{X}))}$$

Note that if $\mathsf{std}(P)$ then $\mathsf{fsh}[m](P)$ for every $m \in \mathcal{K}$. Let R_{P} be the set of rules for the predicates std and $\mathsf{fsh}[m]$ for all $m \in \mathcal{K}$.

We define how to transform rules of type (I), (II) and (III) into rules in simple *path* format that can be easily reversed.

Definition 3.1. *For every operator f in Σ_{S}, every static rule of type (I) for f is converted into*

$$\textit{(1)} \quad \frac{\{X_i \stackrel{a_i[m]}{\rightarrow} X_i'\}_{i \in I} \quad \{\mathsf{std}(X_e)\}_{e \in E} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S \setminus I}}{f(\overrightarrow{X}) \stackrel{a[m]}{\rightarrow} f(\overrightarrow{X'})}$$

where $X_i' = X_i$ for all $i \notin I$. The reverse version is

$$\textit{(1R)} \quad \frac{\{X_i \stackrel{a_i[m]}{\rightsquigarrow} X_i'\}_{i \in I} \quad \{\mathsf{std}(X_e)\}_{e \in E} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S \setminus I}}{f(\overrightarrow{X}) \stackrel{a[m]}{\rightsquigarrow} f(\overrightarrow{X'})}$$

Note that (1) and (1R) are rule schemas for keys m. Also, $I \cap E = \emptyset$, and so predicates only apply to inactive arguments. This contributes to making our rules easily reversible. Finally note that we shall be able to prove that if $P \stackrel{a[m]}{\rightarrow} P'$ then $\mathsf{fsh}[m](P)$ (Lemma 5.2).

Definition 3.2. *For every operator f in Σ_{S}, every choice rule of type (II) for f is converted into*

$$\textit{(2)} \quad \frac{X_d \stackrel{a[m]}{\rightarrow} X_d' \quad \{\mathsf{std}(X_e)\}_{e \in E \setminus \{d\}} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S}}{f(\overrightarrow{X}) \stackrel{a[m]}{\rightarrow} f(\overrightarrow{X'})}$$

where $X_i' = X_i$ for all $i \neq d$. The reverse version of (2) is

$$\textit{(2R)} \quad \frac{X_d \stackrel{a[m]}{\rightsquigarrow} X_d' \quad \{\mathsf{std}(X_e)\}_{e \in E \setminus \{d\}} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S}}{f(\overrightarrow{X}) \stackrel{a[m]}{\rightsquigarrow} f(\overrightarrow{X'})}$$

Again (2) and (2R) are rule schemas for keys m, and again predicates are only applied to inactive arguments, since $d \notin S$.

In order to make operators f with rules of type (III) static we shall use auxiliary operators. These operators have their own rules (type (3′) below) which propagate the actions of a single argument leaving other arguments unchanged.

Definition 3.3. *For every operator f in Σ_{S}, every rule r of type (III) for f is converted into the rule schemas below for all $b \in \mathsf{Act}$ and keys m, n:*

$$(3) \quad \frac{\{\mathsf{std}(X_e)\}_{e \in E} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S}}{f(\overrightarrow{X}) \stackrel{\mathsf{act}(r)[m]}{\longrightarrow} f_r[m](\overrightarrow{X})}$$

$$(3') \quad \frac{X_{\mathsf{ta}(r)} \stackrel{b[m]}{\longrightarrow} X'_{\mathsf{ta}(r)} \quad \{\mathsf{std}(X_e)\}_{e \in E \setminus \{\mathsf{ta}(r)\}} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S}}{f_r[n](\overrightarrow{X}) \stackrel{b[m]}{\longrightarrow} f_r[n](\overrightarrow{X'})} \quad m \neq n$$

The reverse versions of rule schemas of type (3) and (3′) are

$$(3R) \quad \frac{\{\mathsf{std}(X_e)\}_{e \in E} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S}}{f_r[m](\overrightarrow{X}) \stackrel{\mathsf{act}(r)[m]}{\rightsquigarrow} f(\overrightarrow{X})}$$

$$(3'R) \quad \frac{X_{\mathsf{ta}(r)} \stackrel{b[m]}{\rightsquigarrow} X'_{\mathsf{ta}(r)} \quad \{\mathsf{std}(X_e)\}_{e \in E \setminus \{\mathsf{ta}(r)\}} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S}}{f_r[n](\overrightarrow{X}) \stackrel{b[m]}{\rightsquigarrow} f_r[n](\overrightarrow{X'})} \quad m \neq n$$

Again predicates are only applied to inactive arguments.

Now we are ready to define our procedure that reformulates standard operators and produces automatically their new forward and reverse rules. Note that all rules mentioned in Definitions 3.1, 3.2 and 3.3 are in the simple *path* format.

Definition 3.4 (Conversion Procedure). *A simple process calculus $L_{\mathrm{S}} = (\Sigma_{\mathrm{S}}, \mathsf{Act}, R_{\mathrm{S}})$ generates a* reversible process calculus with communication keys *$L = (\Sigma_{\mathrm{SA}}, \mathsf{ActK}, R_{\mathrm{F}}, R_{\mathrm{R}})$ as follows:*

1. *$\Sigma_{\mathrm{SA}} \stackrel{\mathrm{df}}{=} \Sigma_{\mathrm{S}} \cup \Sigma_{\mathrm{A}}$. The operators in Σ_{SA} are called* reversible *operators.*
2. *The forward rule set R_{F} is the least set such that*
   (a) *$R_{\mathrm{P}} \subseteq R_{\mathrm{F}}$, where R_{P} is the set of rules for predicates defined above;*
   (b) *for every rule $r \in R_{\mathrm{S}}$ for f of type (I) or (II) the set R_{F} contains the converted rules r' of the corresponding type (1) or (2) as required by Definitions 3.1 and 3.2;*
   (c) *for every rule $r \in R_{\mathrm{S}}$ for f of type (III) the set R_{F} contains the converted rule r' of type (3), and all the rules of type (3′) for the auxiliary operators $f_r[m]$ as required by Definition 3.3.*
3. *The reverse rule set R_{R} is defined like R_{F}, except that we use the reverse forms of the rules as in Definitions 3.1, 3.2 and 3.3.*

Once L is generated by the procedure in Definition 3.4, we associate with L, in the standard way [3], the forward and reverse ltrs $\rightarrow$ and $\rightsquigarrow$ over Proc with labels drawn from ActK, together with the set of predicates Pred that interpret std and $\mathsf{fsh}[m]$ (for $m \in \mathcal{K}$) over Proc.

We illustrate the application of the conversion procedure on two operators that use the three allowed types of rules. Firstly, we consider the internal choice "$\sqcap$" of CSP, which may be defined by two choice axioms ($\tau \in \mathsf{Act}$):

$$\frac{}{X \sqcap Y \xrightarrow{\tau}_{\mathrm{S}} X} \qquad \frac{}{X \sqcap Y \xrightarrow{\tau}_{\mathrm{S}} Y}$$

Arguments X and Y both belong to E. Definition 3.3 requires two families of auxiliary operators "$\sqcap_1[m]$" and "$\sqcap_2[m]$" for all $m \in \mathcal{K}$. To save space, we only give the converted rules and the reverse rules for the first argument X:

$$\frac{\mathsf{std}(X) \quad \mathsf{std}(Y)}{X \sqcap Y \xrightarrow{\tau[m]} X \sqcap_1 [m] Y} \qquad \frac{X \xrightarrow{a[n]} X' \quad \mathsf{std}(Y)}{X \sqcap_1 [m] Y \xrightarrow{a[n]} X' \sqcap_1 [m] Y} \quad m \neq n$$

$$\frac{\mathsf{std}(X) \quad \mathsf{std}(Y)}{X \sqcap_1 [m] Y \overset{\tau[m]}{\rightsquigarrow} X \sqcap Y} \qquad \frac{X \overset{a[n]}{\rightsquigarrow} X' \quad \mathsf{std}(Y)}{X \sqcap_1 [m] Y \overset{a[n]}{\rightsquigarrow} X' \sqcap_1 [m] Y} \quad m \neq n$$

Next, we convert Milner's interrupt operator "$\hat{}$" [16] defined by the first two rule schemas below (all $a, b \in \mathsf{Act}$). We have $S = I = \{X\}$, $D = \{Y\}$, $E = D$ and Y is permissive. Definitions 3.1 and 3.2 give us the last two forward rule schemas below, and the reverse rules are simply symmetric versions of the forward rules.

$$\frac{X \xrightarrow{a}_{\mathrm{S}} X'}{X \,\hat{}\, Y \xrightarrow{a}_{\mathrm{S}} X' \,\hat{}\, Y} \quad \frac{Y \xrightarrow{b}_{\mathrm{S}} Y'}{X \,\hat{}\, Y \xrightarrow{b}_{\mathrm{S}} Y'} \quad \frac{X \xrightarrow{a[m]} X' \quad \mathsf{std}(Y)}{X \,\hat{}\, Y \xrightarrow{a[m]} X' \,\hat{}\, Y} \quad \frac{Y \xrightarrow{b[n]} Y' \quad \mathsf{fsh}[n](X)}{X \,\hat{}\, Y \xrightarrow{b[n]} X \,\hat{}\, Y'}$$

4 CCS with Communication Keys

In this section we convert CCS to a reversible process calculus, which we call CCSK (CCS with communication Keys), following Definition 3.4. Let $\tau \in \mathsf{Act}$. We assume the following standard signature of finite CCS:

$$\Sigma_{\mathrm{S}} = \{\mathbf{0}\} \cup \{a. \mid a \in \mathsf{Act}\} \cup \{\, \backslash A, [f] \mid A \subseteq \mathsf{Act} \setminus \{\tau\}, f : \mathsf{Act} \rightarrow \mathsf{Act}\} \cup \{+, |\}$$

The single argument of prefixing is neither dynamic nor static, and prefixing has a choice axiom rule (type (III)). By Definition 3.3 CCSK contains a family of auxiliary operators $a[m].$ (past action prefixing) for all $a \in \mathsf{Act}$ and $m \in \mathcal{K}$. Both arguments of $+$ are dynamic and permissive, and obviously non-static. Parallel composition, restriction and relabelling are operators with static rules. The well-known SOS rules for CCS, which can be found in [16], are converted into the rules in Figure 1. The rules for the reverse ltr for CCSK are got by simply changing $\rightarrow$ into $\rightsquigarrow$ throughout. As is usual, we omit trailing $\mathbf{0}$s.

$$\frac{\mathsf{std}(X)}{a.X \stackrel{a[m]}{\rightarrow} a[m].X} \qquad \frac{X \stackrel{b[n]}{\rightarrow} X'}{a[m].X \stackrel{b[n]}{\rightarrow} a[m].X'}\ m \neq n$$

$$\frac{X \stackrel{a[m]}{\rightarrow} X' \quad \mathsf{std}(Y)}{X + Y \stackrel{a[m]}{\rightarrow} X' + Y} \qquad \frac{Y \stackrel{a[m]}{\rightarrow} Y' \quad \mathsf{std}(X)}{X + Y \stackrel{a[m]}{\rightarrow} X + Y'}$$

$$\frac{X \stackrel{a[m]}{\rightarrow} X' \quad \mathsf{fsh}[m](Y)}{X \mid Y \stackrel{a[m]}{\rightarrow} X' \mid Y} \qquad \frac{Y \stackrel{a[m]}{\rightarrow} Y' \quad \mathsf{fsh}[m](X)}{X \mid Y \stackrel{a[m]}{\rightarrow} X \mid Y'} \qquad \frac{X \stackrel{a[m]}{\rightarrow} X' \quad Y \stackrel{\overline{a}[m]}{\rightarrow} Y'}{X \mid Y \stackrel{\tau[m]}{\rightarrow} X' \mid Y'}$$

$$\frac{X \stackrel{a[m]}{\rightarrow} X'}{X \backslash A \stackrel{a[m]}{\rightarrow} X' \backslash A}\ a \notin A \qquad \frac{X \stackrel{a[m]}{\rightarrow} X'}{X[f] \stackrel{f(a)[m]}{\rightarrow} X'[f]}$$

Fig. 1. Forward SOS rules for CCSK

As an extension, we could add recursion rec$X.P$ to CCSK by introducing the structural congruence $\equiv$ generated by the law rec$X.P \equiv P\{\text{rec}X.P/X\}$. We would then add a *Structural Congruence Rule* schema as in [17] to the rules in Figure 1. The schema links structural congruence with deriving transitions of terms: $X \stackrel{a[m]}{\rightarrow} X'$ can be derived if $X \equiv Y$, $Y \stackrel{a[m]}{\rightarrow} Y'$ and $Y' \equiv X'$ for all labels $a[m]$. To incorporate this extension into our format from Sections 2 and 3, we would need to work with formats with structural congruence (cf. [17]).

Example 4.1. In CCSK we keep track of the identities of actions that communicate so that when we reverse we undo the correct past actions. Consider $P = (a.b \mid a.c \mid \overline{a}.d \mid \overline{a}.e)\backslash a$. Here the restriction of a prevents a and $\overline{a}$ being performed except as part of a communication. Suppose that $a.b$ communicates with $\overline{a}.d$ and then $a.c$ with $\overline{a}.e$. In CCSK we write this as follows:

$$P \stackrel{\tau[m]}{\rightarrow} (a[m].b \mid a.c \mid \overline{a}[m].d \mid \overline{a}.e)\backslash a \stackrel{\tau[n]}{\rightarrow} (a[m].b \mid a[n].c \mid \overline{a}[m].d \mid \overline{a}[n].e)\backslash a$$

Note that the process $a[m].b \mid a.c \mid \overline{a}[m].d \mid \overline{a}.e$ cannot regress by reversing $a[m]$ alone because key m is not fresh in $a.c \mid \overline{a}[m].d \mid \overline{a}.e$. The fact that m appears in $a.c \mid \overline{a}[m].d \mid \overline{a}.e$ which is in parallel with $a[m].b$ proves that the processes communicated a and $\overline{a}$.

Our notation does not allow us to backtrack by undoing a different pair of actions, but clearly we can change the order of reversing actions $\tau[m]$ and $\tau[n]$:

$$(a[m].b \mid a[n].c \mid \overline{a}[m].d \mid \overline{a}[n].e)\backslash a \stackrel{\tau[m]}{\rightsquigarrow} (a.b \mid a[n].c \mid \overline{a}.d \mid \overline{a}[n].e)\backslash a \stackrel{\tau[n]}{\rightsquigarrow} P.$$

4.1 Related Calculi

The present work is mainly to be compared with Danos and Krivine's RCCS [9], but also in some sense to an earlier approach by Boudol and Castellani [6].

To aid comparison we give a simple example: the processes $(a \mid \overline{a}.b)\backslash a$ and $\tau.b$. We might reasonably expect them to be equivalent, and indeed they are FR-bisimilar as stated in Section 6. We have $(a \mid \overline{a}.b)\backslash a \stackrel{\tau[m]}{\rightarrow} (a[m] \mid \overline{a}[m].b)\backslash a$ and $\tau.b \stackrel{\tau[m]}{\rightarrow} \tau[m].b$. In RCCS since $\langle\,\rangle \triangleright \nu a\,(a \mid \overline{a}.b) \equiv \nu a\,(\langle 1\rangle \triangleright a \mid \langle 2\rangle \triangleright \overline{a}.b)$ we write these transitions as

$$\nu a\,(\langle 1\rangle \triangleright a \mid \langle 2\rangle \triangleright \overline{a}.b) \stackrel{\langle 1\rangle,\langle 2\rangle:\tau}{\rightarrow} \nu a\,(\langle\langle 2\rangle, a, \mathbf{0}\rangle \cdot \langle 1\rangle \triangleright \mathbf{0} \mid \langle\langle 1\rangle, \overline{a}, \mathbf{0}\rangle \cdot \langle 2\rangle \triangleright b)$$

and $\langle\,\rangle \triangleright \tau.b \stackrel{\langle\,\rangle:\tau}{\rightarrow} \langle *, \tau, \mathbf{0}\rangle \triangleright b$, respectively. In RCCS transition labels contain extra information concerning which threads contribute. As a result it is harder to show that the processes are equivalent. Presumably one would have to abstract away from the thread information.

We might therefore say that, on the spectrum from intensionality to extensionality, the present work is more extensional than RCCS, though we see from the examples in Section 6 that CCSK definitely has a "true concurrency" flavour in terms of which processes it equates.

In [6] Boudol and Castellani developed *event systems*. Similarly to our approach, they keep track of the whole past of a transition by recording past actions and choices that have been made. These are recorded in the syntax of terms and, unlike in our approach, in the transition labels themselves. For example, where we write $(a \mid \overline{a}.b)\backslash a \stackrel{\tau[m]}{\rightarrow} (a[m] \mid \overline{a}[m].b)\backslash a$, in event systems this is $(a \mid \overline{a}.b)\backslash a \stackrel{\backslash a(a,\overline{a})}{\longrightarrow} (\underline{a} \mid \underline{\overline{a}}.b)\backslash a$ and one needs to use additional rules to work out that the action label of the transition is a τ.

5 Properties of the Transition Relations

In this section we establish various properties of the forward and reverse transition relations defined earlier. In particular we show that the forward-reachable processes are closed under reverse transitions (Proposition 5.6); also that the new forward transition relation is in a sense conservative over the standard transition relation (Theorem 5.8).

We start by noting that the reverse transition relation inverts the forward transition relation:

Proposition 5.1. *Let $P, P' \in \mathsf{Proc}$ and $\mu \in \mathsf{ActK}$. Then $P \stackrel{\mu}{\rightarrow} P'$ iff $P' \stackrel{\mu}{\rightsquigarrow} P$.*

Each process has a set of keys. The set $\mathsf{keys}(P)$ of keys occurring in a process $P \in \mathsf{Proc}$ is defined as follows: $\mathsf{keys}(\mathbf{0}) \stackrel{\text{df}}{=} \emptyset$, $\mathsf{keys}(f(\overrightarrow{P})) \stackrel{\text{df}}{=} \bigcup_{i \in N} \mathsf{keys}(P_i)$ and $\mathsf{keys}(f_r[m](\overrightarrow{P})) \stackrel{\text{df}}{=} \{m\} \cup \bigcup_{i \in N} \mathsf{keys}(P_i)$. Clearly $P \in \mathsf{Std}$ iff $\mathsf{keys}(P) = \emptyset$. Also $\mathsf{fsh}[m](P)$ iff $m \notin \mathsf{keys}(P)$.

Any forward transition uses a fresh key:

Lemma 5.2. *Let $P, P' \in \mathsf{Proc}$. If $P \stackrel{a[m]}{\rightarrow} P'$ then $m \notin \mathsf{keys}(P)$ and $\mathsf{keys}(P') = \mathsf{keys}(P) \cup \{m\}$.*

Let $P \rightarrow Q$ iff $P \xrightarrow{\mu} Q$ for some μ. Let $\rightarrow^*$ denote the reflexive and transitive closure of $\rightarrow$.

Definition 5.3. *A process $P \in \mathsf{Proc}$ is* reachable *if it can be reached by a finite sequence of forward transitions from a process in Std, i.e. there is $Q \in \mathsf{Std}$ such that $Q \rightarrow^* P$. Let Rch denote the set of reachable processes.*

It is easy to check that if $P \in \mathsf{Rch}$ and P' is a subterm of P then also $P' \in \mathsf{Rch}$. It follows from Lemma 5.2 that if $P \in \mathsf{Rch}$ then every $\rightarrow$-computation from a process $Q \in \mathsf{Std}$ to P must have length $|\mathsf{keys}(P)|$.

Of course, not every process is reachable. In CCSK, $a.b[m]$ is not reachable. A more interesting example is $a[m].b[n] \mid \overline{b}[n].\overline{a}[m]$. Here the names and keys match up, but there is a causal inconsistency.

A "diamond" confluence property holds for reverse transitions:

Proposition 5.4 (Reverse Diamond Property). *Let $P, Q, R \in \mathsf{Proc}$.*

1. *If $P \overset{a[m]}{\rightsquigarrow} Q$ and $P \overset{b[m]}{\rightsquigarrow} R$ then $a = b$ and $Q = R$.*
2. *If $P \overset{a[m]}{\rightsquigarrow} Q$ and $P \overset{b[n]}{\rightsquigarrow} R$ with $m \neq n$, then there is S such that $Q \overset{b[n]}{\rightsquigarrow} S$ and $R \overset{a[m]}{\rightsquigarrow} S$.*

Proposition 5.4 implies that the reverse transition relation is finitely branching, since the number of reverse transitions of $P \in \mathsf{Proc}$ is bounded by $|\mathsf{keys}(P)|$.

The analogue of Proposition 5.4 does not hold for forward transitions, since two forward transitions $P \xrightarrow{a[m]} Q$ and $P \xrightarrow{b[n]} R$ may conflict. However we can complete the diamond if the forward transitions are compatible, in the sense that Q and R can reach a common process S by forward moves:

Proposition 5.5 (Forward Diamond Property). *Let $P, Q, R, T \in \mathsf{Proc}$.*

1. *If $P \xrightarrow{a[m]} Q \xrightarrow{s} T$ and $P \xrightarrow{b[m]} R \xrightarrow{t} T$ then $a = b$ and $Q = R$.*
2. *If $P \xrightarrow{a[m]} Q \xrightarrow{s} T$ and $P \xrightarrow{b[n]} R \xrightarrow{t} T$ with $m \neq n$, then there is S such that $Q \xrightarrow{b[n]} S$, $R \xrightarrow{a[m]} S$ and $S \xrightarrow{s \backslash b[n]} T$, $S \xrightarrow{t \backslash a[m]} T$.*

(Here for any $s \in \mathsf{ActK}^*$ and $\mu \in \mathsf{ActK}$, $s \backslash \mu$ is s with all instances of μ removed.)

The reachable terms are closed under reverse transitions, meaning that a process can never reverse into an "inconsistent" past:

Proposition 5.6. *If $P \in \mathsf{Rch}$, $\mu \in \mathsf{ActK}$ and $P \overset{\mu}{\rightsquigarrow} P'$ then $P' \in \mathsf{Rch}$.*

We now turn to showing that the new forward transition relation $\rightarrow$ is essentially conservative over the standard transition relation $\rightarrow_{\mathrm{S}}$. We have to take into account the fact that we have introduced auxiliary operators and keys. A nonstandard process can be converted to a corresponding standard process by "pruning" the auxiliary operators (cf. the forgetful map of [9]):

Definition 5.7. *The pruning map* $\pi : \mathsf{Proc} \to \mathsf{Std}$ *is defined as follows:*

$$\pi(\mathbf{0}) \stackrel{\mathrm{df}}{=} \mathbf{0}$$

$$\pi(f(\overrightarrow{P})) \stackrel{\mathrm{df}}{=} \begin{cases} \pi(P_d) & \text{if } d \in D_f \wedge \neg\mathsf{std}(P_d) \wedge \forall e \in E_f \setminus \{d\}.\, \mathsf{std}(P_e) \\ f(\overrightarrow{\pi(P)}) & \text{if } \forall e \in E_f.\, \mathsf{std}(P_e) \\ \mathbf{0} & \text{otherwise}S \end{cases}$$

$$\pi(f_r[m](\overrightarrow{P})) \stackrel{\mathrm{df}}{=} \begin{cases} \pi(P_{\mathsf{ta}(r)}) & \text{if } \forall e \in E_f \setminus \{\mathsf{ta}(r)\}.\, \mathsf{std}(P_e) \\ \mathbf{0} & \text{otherwise} \end{cases}$$

for any choice axiom r *for* f*, and where* $\overrightarrow{\pi(P)}$ *is the vector* $\pi(P_1), \ldots, \pi(P_n)$*.*

Clearly, if $P \in \mathsf{Std}$ then $\pi(P) = P$. It can easily be shown that the third case for $\pi(f(\overrightarrow{P}))$ and the second case for $\pi(f_r(\overrightarrow{P}))$ will not arise with reachable terms.

Theorem 5.8 (Conservation). *Suppose* $P \in \mathsf{Proc}$*.*

1. *If* $P \xrightarrow{a[m]} P'$ *then* $\pi(P) \xrightarrow{a}_{\mathrm{S}} \pi(P')$*.*
2. *If* $\pi(P) \xrightarrow{a}_{\mathrm{S}} P'$ *then for any* $m \in \mathcal{K} \setminus \mathsf{keys}(P)$ *there is* P'' *such that* $P \xrightarrow{a[m]} P''$ *and* $\pi(P'') = P'$*.*

6 Forward-Reverse Bisimulation

We can show that the reversible transition relation $\to$ induces essentially the same bisimulation equivalence on processes as the standard transition relation $\to_{\mathrm{S}}$. We first recall standard strong bisimulation on the standard terms:

Definition 6.1. *A symmetric relation* $\mathcal{S}$ *on* Std *is an* S-bisimulation *if whenever* $\mathcal{S}(P, Q)$ *then if* $P \xrightarrow{a}_{\mathrm{S}} P'$ *then there is* Q' *such that* $Q \xrightarrow{a}_{\mathrm{S}} Q'$ *and* $\mathcal{S}(P', Q')$*. We define* $P \sim_{\mathsf{S}} Q$ *iff there is an S-bisimulation* $\mathcal{S}$ *such that* $\mathcal{S}(P, Q)$*.*

The corresponding notion for forward transitions on Proc and predicates Pred is

Definition 6.2. *A symmetric relation* $\mathcal{S}$ *on* Proc *is an* F-bisimulation *if* $\mathcal{S}(P, Q)$ *implies*

- $\mathsf{p}(P) \Leftrightarrow \mathsf{p}(Q)$ *for all* $\mathsf{p} \in \mathsf{Pred}$*;*
- *if* $P \xrightarrow{\mu} P'$ *then there is* Q' *such that* $Q \xrightarrow{\mu} Q'$ *and* $\mathcal{S}(P', Q')$*.*

We define $P \sim_{\mathsf{F}} Q$ *iff there is an F-bisimulation* $\mathcal{S}$ *such that* $\mathcal{S}(P, Q)$*.*

Note that the first item in Definition 6.2 could be written as $\mathsf{keys}(P) = \mathsf{keys}(Q)$, since $\mathsf{fsh}[m](P) \Leftrightarrow m \notin \mathsf{keys}(P)$ and $\mathsf{std}(P) \Leftrightarrow \mathsf{keys}(P) = \emptyset$.

F-bisimulation is conservative over S-bisimulation by the following result:

Proposition 6.3. *Let* $P, Q \in \mathsf{Proc}$*. Then* $P \sim_{\mathsf{F}} Q$ *iff* $\pi(P) \sim_{\mathsf{S}} \pi(Q)$ *and* $\mathsf{p}(P) \Leftrightarrow \mathsf{p}(Q)$ *for all* $\mathsf{p} \in \mathsf{Pred}$*.*

Proposition 6.4. *The relation* $\sim_{\mathsf{F}}$ *is a congruence with respect to all the operators of* Proc*.*

We now define bisimulation for both forward and reverse transitions:

Definition 6.5. *A symmetric relation $\mathcal{S}$ on* Proc *is a* forward-reverse (FR) bisimulation *if whenever $\mathcal{S}(P,Q)$ then*

- $\mathsf{p}(P) \Leftrightarrow \mathsf{p}(Q)$ *for all* $\mathsf{p} \in \mathsf{Pred}$*;*
- *if $P \xrightarrow{\mu} P'$ then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $\mathcal{S}(P',Q')$;*
- *if $P \overset{\mu}{\rightsquigarrow} P'$ then there is Q' such that $Q \overset{\mu}{\rightsquigarrow} Q'$ and $\mathcal{S}(P',Q')$.*

We define $P \sim_{\mathsf{FR}} Q$ iff there is an FR bisimulation $\mathcal{S}$ such that $\mathcal{S}(P,Q)$.

Proposition 6.6. *Let $P, Q \in \mathsf{Proc}$. If $P \sim_{\mathsf{FR}} Q$ then $P \sim_{\mathsf{F}} Q$.*

The converse does not hold. For instance we have $a \mid a \sim_{\mathsf{F}} a.a$, but $a \mid a \not\sim_{\mathsf{FR}} a.a$. This is because $a \mid a \xrightarrow{a[m]}\xrightarrow{a[n]} a[m] \mid a[n] \overset{a[m]}{\rightsquigarrow} a \mid a[n]$ and $m \neq n$. This sequence of transitions cannot be matched by $a.a$: we have $a.a \xrightarrow{a[m]}\xrightarrow{a[n]} a[m].a[n] \overset{a[m]}{\not\rightsquigarrow}$. Similarly $a \mid b \sim_{\mathsf{F}} a.b + b.a$, but $a \mid b \not\sim_{\mathsf{FR}} a.b + b.a$.

On the positive side, we can show that for any $P \in \mathsf{Std}$, $P + P \sim_{\mathsf{FR}} P$. We can also show that for any $P \in \mathsf{Std}$, $(a \mid \overline{a}.P)\backslash a \sim_{\mathsf{FR}} \tau.(P\backslash a)$.

Theorem 6.7. *The relation $\sim_{\mathsf{FR}}$ is a congruence with respect to all the operators of* Proc.

Several notions of bisimulation taking into account backward as well as forward moves have been discussed in the literature. The *back and forth bisimulation* of [11] is constrained to only go back along the path that brought a process to its current state. Back and forth bisimulation where any reverse path can be followed is discussed in [5] both for transition systems and event structures. Essentially the same notion, but called *backward-forward bisimulation*, is defined in [13] for occurrence transition systems. The non-interleaving semantics community has proposed several bisimulation-like equivalences [12] and we intend to investigate how FR bisimulation compares with them.

7 Extensions

Our conversion procedure can be extended in several directions so that it applies to a wider class of operators. Naturally, this would result in extending the forms of SOS rules in Definitions 3.1–3.3. However, the extensions we now briefly describe mostly do not go beyond the simple *path* format as in Definition 2.1.

ACP action constants can be defined analogously to prefixing of CCS. We have the constant ε (successful termination) and constants a for each $a \in \mathsf{Act}$. The defining rules $a \xrightarrow{a}_{\mathrm{S}} \varepsilon$ are converted to $a \xrightarrow{a[m]} a[m]$, where $a[m]$ are auxiliary constants for all $m \in \mathcal{K}$. There are no forward SOS rules for the auxiliary constants and no transition rules for ε.

The next extension is to allow predicates in SOS rules. An example is the successful termination predicate trm in the rules for ACP's sequential composition

"$\cdot$" below [4]. Care needs to be taken when adding predicates to premises in order to avoid *lookahead* in the reverse rules.

$$\frac{X \xrightarrow{a}_{\mathrm{S}} X'}{X \cdot Y \xrightarrow{a}_{\mathrm{S}} X' \cdot Y} \qquad \frac{Y \xrightarrow{b}_{\mathrm{S}} Y' \quad \mathsf{trm}(X)}{X \cdot Y \xrightarrow{b}_{\mathrm{S}} Y'}$$

With some simplifications, the converted and reverse rules are

$$\frac{X \xrightarrow{a} X' \quad \mathsf{std}(Y)}{X \cdot Y \xrightarrow{a} X' \cdot Y} \quad \frac{Y \xrightarrow{b} Y' \quad \mathsf{trm}(X)}{X \cdot Y \xrightarrow{b} X \cdot Y'} \quad \frac{X \stackrel{a}{\rightsquigarrow} X' \quad \mathsf{std}(Y)}{X \cdot Y \stackrel{a}{\rightsquigarrow} X' \cdot Y} \quad \frac{Y \stackrel{b}{\rightsquigarrow} Y' \quad \mathsf{trm}(X)}{X \cdot Y \stackrel{b}{\rightsquigarrow} X \cdot Y'}$$

(Here we extend trm to cover nonstandard processes.)

Finally, to allow the external choice operator of CSP we need to relax the condition that static arguments cannot be dynamic. The defining rules for "$\Box$" are given below, where the last two rules are rule schemas for all $a \in \mathsf{Act} \setminus \{\tau\}$.

$$\frac{X \xrightarrow{\tau}_{\mathrm{S}} X'}{X \Box Y \xrightarrow{\tau}_{\mathrm{S}} X' \Box Y} \quad \frac{Y \xrightarrow{\tau}_{\mathrm{S}} Y'}{X \Box Y \xrightarrow{\tau}_{\mathrm{S}} X \Box Y'} \quad \frac{X \xrightarrow{a}_{\mathrm{S}} X'}{X \Box Y \xrightarrow{a}_{\mathrm{S}} X'} \quad \frac{Y \xrightarrow{a}_{\mathrm{S}} Y'}{X \Box Y \xrightarrow{a}_{\mathrm{S}} Y'}$$

By introducing an auxiliary predicate $\mathsf{before}(P)$, which holds if $P \in \mathsf{Std}$ or P is a derivative from a standard term via a sequence of silent actions, we obtain the following converted rules:

$$\frac{\mathsf{before}(Y) \qquad X \xrightarrow{\mu} X'}{X \Box Y \xrightarrow{\mu} X' \Box Y} \qquad \frac{\mathsf{before}(X) \qquad Y \xrightarrow{\mu} Y'}{X \Box Y \xrightarrow{\mu} X \Box Y'}.$$

8 Conclusions

There has been much recent interest in reversible computing, including the pioneering work of Danos and Krivine on reversible CCS. We have introduced a method for converting standard irreversible operators of algebraic process calculi such as CCS into reversible operators. As far as we are aware, this is the first time that such a method has been proposed in the context of general process calculi. Our method works on operators with rules of a simple form. We arrive at new rules which preserve the structure of the terms. An important feature of our method is the introduction of keys to bind synchronised actions together. We have also obtained an appropriate notion of bisimulation on terms. Our work demonstrates that it is possible to make many standard operators reversible in a manner which is both algebraic and tractable.

Acknowledgements

We wish to thank Philippa Gardner, Daniele Varacca, Nobuko Yoshida, Shoji Yuen and the referees for helpful discussions and comments. The second author would like to thank the University of Leicester for granting study leave, and acknowledge gratefully support from Nagoya University during a research visit.

References

[1] S. Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005.

[2] T. Altenkirch and J. Grattage. A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005*, pages 249–258. IEEE Computer Society Press, 2005.

[3] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In *Proceedings of 4th International Conference on Concurrency Theory, CONCUR '93*, volume 715 of *LNCS*, pages 477–492. Springer-Verlag, 1993.

[4] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[5] M.A. Bednarczyk. Hereditary history preserving bisimulations or what is the power of the future perfect in program logics. Technical report, Institute of Computer Science, Polish Academy of Sciences, Gdańsk, 1991.

[6] G. Boudol and I. Castellani. Flow models of distributed computations: three equivalent semantics for CCS. *Information and Computation*, 114:247–314, 1994.

[7] H. Buhrman, J. Tromp, and P. Vitányi. Time and space bounds for reversible simulation. In *Proceedings of 28th International Colloquium on Automata, Languages and Programming, ICALP 2001*, volume 2076 of *LNCS*, pages 1017–1027. Springer-Verlag, 2001.

[8] V. Danos and J. Krivine. Formal molecular biology done in CCS-R. In *Proceedings of BioConcur, Marseille*, 2003.

[9] V. Danos and J. Krivine. Reversible communicating systems. In *Proceedings of the 15th International Conference on Concurrency Theory, CONCUR 2004*, volume 3170 of *LNCS*, pages 292–307. Springer-Verlag, 2004.

[10] V. Danos and J. Krivine. Transactions in RCCS. In *Proceedings of the 16th International Conference on Concurrency Theory, CONCUR 2005*, volume 3653 of *LNCS*, pages 398–412. Springer-Verlag, 2005.

[11] R. De Nicola, U. Montanari, and F. Vaandrager. Back and forth bisimulations. In *Proceedings of CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, pages 152–165. Springer-Verlag, 1990.

[12] R.J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37:229–327, 2001.

[13] U. Goltz, R. Kuiper, and W. Penczek. Propositional temporal logics and equivalences. In *Proceedings of 3rd International Conference on Concurrency Theory, CONCUR '92*, volume 630 of *LNCS*, pages 222–235. Springer-Verlag, 1992.

[14] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[15] R. Landauer. Irreversibility and heat generated in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[16] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[17] M.R. Mousavi and M.A. Reniers. Congruence for structural congruences. In *Proceedings of the 8th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2005*, volume 3441 of *LNCS*, pages 47–62. Springer-Verlag, 2005.

[18] I.C.C. Phillips and I. Ulidowski. Reversing algebraic process calculi. Technical Report CS-06-01, Department of Computer Science, Leicester University, 2006.

[19] G.D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[20] Virtutech. *Simics Hindsight*. http://www.virtutech.com, 2005.

Conjunction on Processes: Full–Abstraction Via Ready–Tree Semantics

Gerald Lüttgen[1,⋆] and Walter Vogler[2]

[1] Department of Computer Science, University of York,
York YO10 5DD, UK
luettgen@cs.york.ac.uk
[2] Institut für Informatik, Universität Augsburg,
D–86135 Augsburg, Germany
vogler@informatik.uni-augsburg.de

Abstract. A key problem in mixing operational (e.g., process–algebraic) and declarative (e.g., logical) styles of specification is how to deal with inconsistencies arising when composing processes under conjunction. This paper introduces a conjunction operator on labelled transition systems capturing the basic intuition of "*a and b = false*", and considers a naive preorder that demands that an inconsistent specification can only be refined by an inconsistent implementation.

The main body of the paper is concerned with characterising the largest precongruence contained in the naive preorder. This characterisation will be based on a novel semantics called ready–tree semantics, which refines ready traces but is coarser than ready simulation. It is proved that the induced ready–tree preorder is compositional and fully–abstract, and that the conjunction operator indeed reflects conjunction.

The paper's results provide a foundation for, and an important step towards a unified framework that allows one to freely mix operators from process algebras and temporal logics.

1 Introduction

Process algebra [2] and *temporal logic* [14] are two popular approaches to formally specifying and reasoning about reactive systems. The process–algebraic paradigm is founded on notions of *refinement*, where one typically formulates a system specification and its implementation in the same notation and then proves using *compositional reasoning* that the latter refines the former. The underlying semantics is often given operationally, and refinement relations are formalised as precongruences. In contrast, the temporal–logic paradigm is based on the use of temporal logics to formulate specifications abstractly, with implementations being denoted in an operational notation. One then verifies a system by establishing that it is a model of its specification.

⋆ Research support was partially provided by the NSF under grant CCR–9988489.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 261–276, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Recently, two papers have been published aimed at marrying process algebras and temporal logics [5, 6]. While the first paper introduces a semantic framework based on Büchi automata, the second paper considers labelled transition systems augmented with an "unimplementability predicate". This predicate captures *inconsistencies* arising when composing processes conjunctively; e.g., the composition $a \wedge b$ is contradictory since a run of a process cannot begin with both actions a and b. Moreover, the frameworks in [5, 6] are equipped with a refinement preorder based on De Nicola and Hennessy's must–testing preorder [12]. However, the obtained results are unsatisfactory: the refinement preorder in [5] is not a precongruence, while the $\wedge$–operator in [6] is not *conjunction* with respect to the studied precongruence.

This paper solves the deficiencies of [5, 6] within a simple setting of labelled transition systems in which a state represents either external (non–deterministic) or internal (disjunctive) choice. Moreover, states that are vacuously *true* or *false* are tagged accordingly. The tagging of *false* states, or inconsistent states, is given by an inductive *inconsistency predicate* that is defined very similar but subtly different to the unimplementability predicate of [6]. We then equip our setting with two operators: the conjunction operator $\wedge$ is in essence a synchronous composition on observable actions and an interleaving product on the unobservable action τ, but additionally captures inconsistencies; the disjunction operator $\vee$ simply resembles the process–algebraic operator of internal choice.

Our variant of labelled transition systems gives rise to a naive refinement preorder $\sqsubseteq_F$ requiring that an inconsistent specification cannot be refined except by an inconsistent implementation. We characterise the *consistency preorder*, i.e. the largest precongruence contained in $\sqsubseteq_F$ when conjunctively closing under all contexts. To do so, we define a novel semantics, called *ready–tree semantics* which is — at least when disallowing divergent behaviour — finer than both must–testing semantics [12] and ready–trace semantics [7], but coarser than ready simulation [3]. The resulting *ready–tree preorder* $\lesssim$ is not only compositional for $\wedge$ and $\vee$ and fully–abstract with respect to $\sqsubseteq_F$, but also possesses several other desired properties. In particular, we prove that $\wedge$ ($\vee$) is indeed conjunction (disjunction) relative to $\lesssim$, and that $\wedge$ and $\vee$ satisfy the expected boolean laws, such as the distributivity laws.

Our results are a significant first step towards the goal of developing a uniform calculus in which one can freely mix process–algebraic and temporal–logic operators. This will give engineers powerful tools to model system components at different levels of abstraction and to impose logical constraints on the execution behaviour of components. The proposed ready–tree preorder will allow engineers to step–wise and component–wise refine systems by trading off logical content for operational content.

Organisation. The next section presents our setting of labelled transition systems augmented with *true* and *false* predicates, together with a conjunction and a disjunction operator. Sec. 3 defines the novel ready–tree semantics, addresses expressiveness issues of several ready–tree variants and introduces the ready–tree preorder. Our compositionality and full–abstraction results are stated in Sec. 4.

All proofs can be found in a technical report [11]. Finally, Sec. 5 discusses our results in light of related work, while Sec. 6 presents our conclusions and suggests directions for future research.

2 Labelled Transition Systems and Conjunction

This section first introduces our process–algebraic setting and particularly conjunctive composition informally, discusses semantic choices and their implications, and finally gives a formal account of our framework.

Motivation. Our setting models processes as labelled transition systems, which may be composed conjunctively and disjunctively. As usual in process algebra, transition labels are actions taken from some alphabet $\mathcal{A} = \{a, b, \ldots\}$. When an action a is offered by the environment and the process under consideration is in a state having one or more outgoing a–transitions, the process must choose and perform one of them. If there is no outgoing a–transition, then the process stays in its state, at least in classical process–algebraic frameworks where the composition between a process and its environment is modelled using some parallel operator. However, in a conjunctive setting we wish to mark the composed state between process and environment as inconsistent, if the environment offers an action that the process cannot perform, or vice versa. Hence, taking ordinary synchronous composition as operator for conjunction is insufficient.

We illustrate this intuition behind our conjunction operator $\wedge$ and its implications by the example labelled transition systems of Fig. 1. First, consider the processes p, q and r. Process p and q specify that exactly action a and respectively action b is offered initially. Similarly, process r specifies that a and b are offered initially. From this perspective, $p \wedge q$ as well as $p \wedge r$ are *inconsistent* and should be tagged as such. Formally, our labelled transition systems will be augmented by an *inconsistency predicate* F, so that $p \wedge q,\ p \wedge r \in F$ in our example. We also refer to inconsistent states as *false*–states.

Now consider the conjunction $p' \wedge q'$ shown on the right in Fig. 1. Since both conjuncts require action a to be performed, $p' \wedge q'$ should have an a–transition. From the preceding discussion, this transition should lead to a *false*–state. No sensible process can meet these requirements of being able to perform a and being inconsistent afterwards. Thus, our inconsistency predicate will propagate backwards to the conjunction itself, as indicated in Fig. 1.

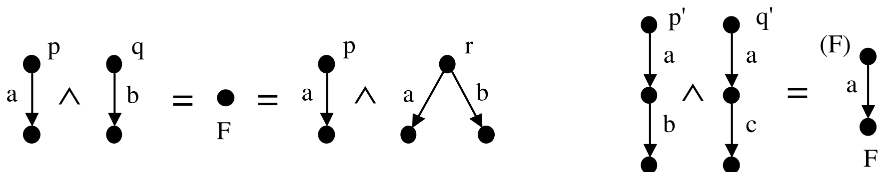


Fig. 1. Basic intuition behind conjunctive composition

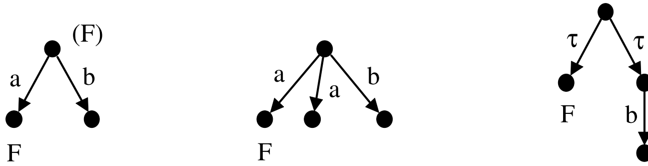


Fig. 2. Backward propagation of inconsistencies

Fig. 2 shows more intricate examples of backward propagation. The inconsistency of the target state of the a–transition of the process on the left propagates backwards to its source state. This is the case although the source state is able to offer a transition leading to a consistent state. However, that transition can only be taken if the environment offers action b. The process is forced into the inconsistency when the environment offers action a.

The situation is different for the process in the middle, which has an additional a–transition leading to a consistent state. Here, the process is consistent, as it can choose to execute this new a–transition and thus avoid to enter a *false*–state. In fact, this choice can be viewed as a disjunction between the two a–branches. As an aside, note that in [6] the design decision was to consider a process already as inconsistent if some a–derivative is. While there might be an intuitive justification for that, it led to a setting where the implied conjunction operator does not reflect conjunction for the studied refinement preorder, i.e., where Thm. 20(1) does not hold.

Disjunction can be made explicit by using the classical *internal–choice* operator. This operator may as usual be expressed by employing the special, unobservable action $\tau \notin \mathcal{A}$ as shown on the right in Fig. 2. Hence, we may identify the internal–choice operator with the disjunction operator $\vee$ desired in our setting. Moreover, a disjunction $p \vee q$ is inconsistent if both p and q are *false*–states. In particular, the process on the right in Fig. 2 will represent $\mathit{false} \vee q$ in our approach, with q from Fig. 1, which clearly should be consistent.

Formalisation. For notational convenience we denote $\mathcal{A} \cup \{\tau\}$ by $\mathcal{A}_\tau$ and use $\alpha, \beta, \ldots$ as representatives of $\mathcal{A}_\tau$. We start off be defining our notion of *labelled transition systems* (LTS). The LTSs considered here are augmented with a *false*–predicate F on states, as discussed above, and dually with a *true*–predicate T. A state in F represents inconsistent, empty behaviour, while a state in T represents completely underspecified, arbitrary behaviour.

Formally, an LTS is a quadruple $\langle P, \longrightarrow, T, F\rangle$, where P is the set of *processes* (states), $\longrightarrow \subseteq P \times \mathcal{A}_\tau \times P$ is the *transition relation*, and $T \subseteq P$ and $F \subseteq P$ is the *true–predicate* and the *false–predicate*, respectively. We write $p \stackrel{\alpha}{\longrightarrow} p'$ instead of $\langle p, \alpha, p'\rangle \in \longrightarrow$, $p \stackrel{\alpha}{\longrightarrow}$ instead of $\exists p' \in P.\, p \stackrel{\alpha}{\longrightarrow} p'$, and $p \longrightarrow$ instead of $\exists p' \in P,\ \alpha \in \mathcal{A}_\tau .\, p \stackrel{\alpha}{\longrightarrow} p'$. When $p \stackrel{\alpha}{\longrightarrow} p'$, we say that process p can perform an α–step to p', and we call p' an α–derivative. We also require an LTS to satisfy the following *τ–purity* condition: $p \stackrel{\tau}{\longrightarrow}$ implies $\nexists a \in \mathcal{A}.\, p \stackrel{a}{\longrightarrow}$, for all $p \in P$. Hence, each process represents either an external or internal (disjunctive) choice between its outgoing transitions. This restriction turns out to be technically

convenient, and we leave exploring the consequences of lifting it for future work. The LTSs of interest to us need to satisfy four further properties, as stated in the following formal definition:

Definition 1 (Logical LTS). An LTS $\langle P, \longrightarrow, T, F\rangle$ is a *logical LTS* if it satisfies the following conditions:

1. $T \cap F = \emptyset$
2. $T \subseteq \{p \,|\, p \not\longrightarrow\}$
3. $F \subseteq P$ such that $p \in F$ whenever $\exists \alpha \in \mathcal{I}(p)\, \forall p' \in P.\; p \stackrel{\alpha}{\longrightarrow} p' \implies p' \in F$
4. p cannot stabilise $\implies p \in F$.

Naturally, we require that a process cannot be tagged *true* and *false* at the same time. As a *true*–process specifies arbitrary, full behaviour, any behaviour made explicit by outgoing transitions is already included implicitly; hence, any outgoing transitions may simply be cut off. The third condition formalises the backwards propagation of inconsistencies as discussed in the motivation section above; here, $\mathcal{I}(p)$ stands for the set $\{\alpha \in \mathcal{A}_\tau \,|\, p \stackrel{\alpha}{\longrightarrow}\}$ of initial actions of process p, to which we also refer as *ready set*.

The fourth condition relates to *divergence*, i.e., infinite sequences of τ–transitions. In many semantic frameworks, e.g. [12, 7], divergence is considered catastrophic, while in our setting catastrophic behaviour is inconsistent behaviour. We view divergence only as catastrophic if a process cannot *stabilise*, i.e., if it cannot get out of an infinite, internal computation. While this is intuitive, there is also a technical reason to which we will come back shortly.

To formalise our notion of stabilisation, we first introduce a weak transition relation $\Longrightarrow \subseteq P \times (\mathcal{A}_\tau \cup \{\epsilon\}) \times P$ which is defined by (1) $p \stackrel{\varepsilon}{\Longrightarrow} p'$ if $p \equiv p' \notin F$, where $\equiv$ denotes syntactic equality, or if $p \notin F$ and $p \stackrel{\tau}{\longrightarrow} p'' \stackrel{\varepsilon}{\Longrightarrow} p'$ for some p'', and (2) $p \stackrel{a}{\Longrightarrow} p'$ if $p \notin F$ and $p \stackrel{a}{\longrightarrow} p'' \stackrel{\varepsilon}{\Longrightarrow} p'$ for some p''. Our definition of a weak transition is slightly unusual: a weak transition cannot pass through *false*–states since these cannot occur in computations, and it does not abstract from τ–transitions preceding a visible transition. However, we only will use weak visible transitions from *stable* states, i.e., states with no outgoing τ–transition. Finally, we can now formalise stabilisation: a process p *can stabilise* if $p \stackrel{\varepsilon}{\Longrightarrow} p'$ for some stable p'.

Note that both Conds. (3) and (4) are inductively defined conditions. We refer to them as *fixed point conditions of F for LTS* . For convenience, we will often write LTS instead of logical LTS in the sequel. Moreover, whenever we mention a process p without stating a respective LTS explicitly, we assume implicitly that such an LTS $\langle P, \longrightarrow, T, F\rangle$ is given. We let *tt* (*ff*) stand for the *true* (*false*) process, which is the only process of an LTS with $tt \in T$ ($ff \in F$).

Operators. Our conjunction operator is essentially a synchronous composition for visible transitions and an asynchronous composition for τ–transitions. However, we need to take care of the T– and F–predicates.

Definition 2 (Conjunction Operator). The conjunction of two logical LTSs $\langle P, \longrightarrow_P, T_P, F_P\rangle$, $\langle Q, \longrightarrow_Q, T_Q, F_Q\rangle$ is the LTS $\langle P \wedge Q, \longrightarrow_{P\wedge Q}, T_{P\wedge Q}, F_{P\wedge Q}\rangle$ defined by:

- $P \wedge Q =_{\mathrm{df}} \{p \wedge q \,|\, p \in P,\, q \in Q\}$
- $\longrightarrow_{P \wedge Q}$ is determined by the following operational rules:

$$\begin{array}{rcl} p \stackrel{\tau}{\longrightarrow}_P p' & \Longrightarrow & p \wedge q \stackrel{\tau}{\longrightarrow}_{P \wedge Q} p' \wedge q \\ q \stackrel{\tau}{\longrightarrow}_Q q' & \Longrightarrow & p \wedge q \stackrel{\tau}{\longrightarrow}_{P \wedge Q} p \wedge q' \\ p \stackrel{a}{\longrightarrow}_P p',\; q \stackrel{a}{\longrightarrow}_Q q' & \Longrightarrow & p \wedge q \stackrel{a}{\longrightarrow}_{P \wedge Q} p' \wedge q' \\ q \in T_Q,\; p \stackrel{a}{\longrightarrow}_P p' & \Longrightarrow & p \wedge q \stackrel{a}{\longrightarrow}_{P \wedge Q} p' \wedge q \\ p \in T_P,\; q \stackrel{a}{\longrightarrow}_Q q' & \Longrightarrow & p \wedge q \stackrel{a}{\longrightarrow}_{P \wedge Q} p \wedge q' \end{array}$$

- $p \wedge q \in T_{P \wedge Q}$ if and only if $p \in T_P$ and $q \in T_Q$
- $p \wedge q \in F_{P \wedge Q}$ if at least one of the following conditions holds:
  1. $p \in F_P$ or $q \in F_Q$
  2. $p \notin T_P$ and $q \notin T_Q$ and $p \wedge q \stackrel{\tau}{\not\longrightarrow}_{P \wedge Q}$ and $\mathcal{I}(P) \neq \mathcal{I}(Q)$
  3. $\exists \alpha \in \mathcal{I}(p \wedge q)\, \forall p' \wedge q'.\, p \wedge q \stackrel{\alpha}{\longrightarrow}_{P \wedge Q} p' \wedge q' \;\Longrightarrow\; p' \wedge q' \in F_{P \wedge Q}$
  4. $p \wedge q$ cannot stabilise

Note that the treatment of *true*–processes when defining $\longrightarrow_{P \wedge Q}$ and $T_{P \wedge Q}$ reflects our intuition that these processes allow arbitrary behaviour. We are left with explaining Conds. (1)–(4). Firstly, a conjunction is inconsistent if any conjunct is. Conds. (2) and (3) reflect our intuition of inconsistency and, respectively, backward propagation stated in the motivation section above. Cond. (4) is added to enforce Def. 1(4). We refer to Conds. (3) and (4) as *fixed point conditions of F for* $\wedge$.

It is easy to check that conjunction is well–defined, i.e., that the conjunctive composition of two logical LTSs satisfies the four conditions of Def. 1. For Def. 1(1) in particular, note that $p \wedge q \in T_{P \wedge Q}$ does not satisfy any of the four conditions for $F_{P \wedge Q}$.

We may now demonstrate why we have treated non–escapable divergence as catastrophic in our setting. This is because, otherwise, our conjunction operator would not be associative as demonstrated by the example depicted in Fig. 3. If the conjunction is computed from the left, the result is the first conjunct. Computed from the right, the result is the same but with both processes being in F. Hence, in the first case, the divergence hides the inconsistency. Since this is not really plausible and associativity of conjunction is clearly desirable, we need some restriction for divergence; it turns out that restricting divergence to escapable divergence, i.e., potential stabilisation, is sufficient for our purposes.

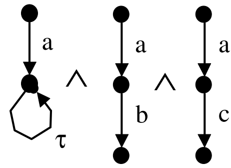


Fig. 3. Counter–example

Definition 3 (Disjunction Operator). The disjunction of two logical LTSs $\langle P, \longrightarrow_P, T_P, F_P \rangle$ and $\langle Q, \longrightarrow_Q, T_Q, F_Q \rangle$ satisfying (w.l.o.g.) $P \cap Q = \emptyset$, is the logical LTS $\langle P \vee Q, \longrightarrow_{P \vee Q}, T_{P \vee Q}, F_{P \vee Q} \rangle$ defined by:

- $P \vee Q =_{\text{df}} \{p \vee q \,|\, p \in P,\, q \in Q\} \cup P \cup Q$
- $\longrightarrow_{P \vee Q}$ is determined by the following operational rules:

$$\begin{array}{rcl} \text{always} & \Longrightarrow & p \vee q \xrightarrow{\tau}_{P\vee Q} p \\ \text{always} & \Longrightarrow & p \vee q \xrightarrow{\tau}_{P\vee Q} q \\ p \xrightarrow{\alpha}_P p' & \Longrightarrow & p \xrightarrow{\alpha}_{P\vee Q} p' \\ q \xrightarrow{\alpha}_Q q' & \Longrightarrow & q \xrightarrow{\alpha}_{P\vee Q} q' \end{array}$$

- $p \vee q \notin T_{P\vee Q}$ always
- $p \vee q \in F_{P\vee Q}$ if and only if $p \in F_P$ and $q \in F_Q$

The definition of disjunction, which reflects internal choice, is quite straightforward and well–defined. Only the definition of $T_{P\vee Q}$ for $p \vee q$ is unusual, as one would expect to simply have $p \vee q \in T$ whenever p or q is in T. However, then Cond. (2) of Def. 1 would be violated. Our alternative definition respects this condition and is semantically equivalent. In the sequel we leave out indices of relations and predicates whenever the context is clear.

Refinement Preorder. As the basis for our semantical considerations we now define a naive refinement preorder stating that an inconsistent specification cannot be implemented except by an inconsistent implementation.

Definition 4 (Naive Consistency Preorder). The *naive consistency preorder* $\sqsubseteq_F$ on processes is defined by $p \sqsubseteq_F q$ if $p \in F \implies q \in F$.

One main objective of this paper is to identify the corresponding fully–abstract preorder with respect to conjunction and disjunction, which is contained in $\sqsubseteq_F$. Our approach follow the testing idea of De Nicola and Hennessy [12], for which we define a testing relation $\sqsubseteq$ as usual. Note that a process and an observer need to be composed not simply synchronously but conjunctively. This is because we want the observer to be sensitive to inconsistencies, so that $p \sqsubseteq q$ if each "conjunctive observer" that sees an inconsistency in p also sees one in q.

Definition 5 (Consistency Testing Preorder). The *consistency testing preorder* $\sqsubseteq$ on processes is defined as the conjunctive closure of the naive consistency preorder under all processes (observers), i.e., $p \sqsubseteq q$ if $\forall o.\, p \wedge o \sqsubseteq_F q \wedge o$.

To characterise the full–abstract precongruence contained in $\sqsubseteq_F$ we will introduce a new semantics, called *ready–tree semantics*, and an associated preorder, the *ready–tree preorder*, which is compositional for conjunction and disjunction and which coincides with $\sqsubseteq$.

Example. As an illustration for our approach, consider process *spec* in Fig. 4. For $\mathcal{A} = \{a, b, c\}$, *spec* specifies that action c can only occur after action a. In the light of the above discussions, an implementation should offer initially either just a, or a and b, or just b, so that *spec* is an internal choice between three states. Moreover, after an action a, nothing more is specified; after an action b, the same

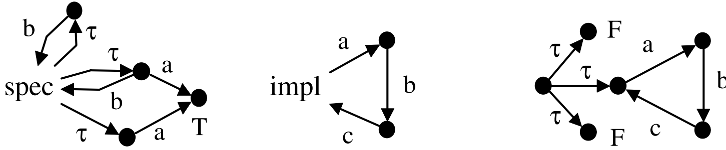


Fig. 4. Example processes

is required as initially. While our specification of this simple behaviour may look quite complex, we may imagine that process *spec* is generated automatically from a temporal–logic formula. Fig. 4 also shows process *impl* which repeats sequence *abc*, and $spec \wedge impl$. It will turn out that $spec \sqsubseteq impl$ (cf. Sec. 4).

3 Ready–Tree Semantics

A first guess for achieving a compositional semantics reflecting consistency testing is to use a kind of *ready–trace semantics* [7]. Such a semantics would refine trace semantics by checking the initial action set of every stable state along each trace. However, this is not sufficient when dealing with inconsistencies, since inconsistencies propagate backwards along traces as explained in Sec. 2. It turns out that a set of tree–like observations is needed, which leads to a novel denotational–style semantics which we call *ready–tree semantics*.

Observation trees & ready trees. A tree–like observation can itself be seen as a deterministic LTS with empty F–predicate, reflecting that observers are internally consistent.

Definition 6 (Observation Tree). An *observation tree* is a LTS $\langle V, \rightarrow, T, \emptyset \rangle$ satisfying the following properties:

1. $\langle V, \longrightarrow \rangle$ is a tree whose root is referred to as v_0
2. $\forall v \in V.\ v$ stable
3. $\forall v \in V, a \in \mathcal{I}(v)\ \exists_1 v' \in V.\ v \overset{a}{\longrightarrow} v'$

We often denote such an observation tree by its root v_0. Next we define the observations of a process p, called *ready trees*. Note that p can only be observed at its stable states.

Definition 7 (Ready Tree). An observation tree v_0 is a *ready tree of* p if there is a labelling $h : V \longrightarrow P$ satisfying the following conditions:

1. $\forall v \in V.\ h(v)$ stable and $h(v) \notin F$
2. $p \overset{\varepsilon}{\Longrightarrow} h(v_0)$
3. $\forall v \in V,\ a \in \mathcal{A}.\ v \overset{a}{\longrightarrow} v'$ implies (a) $h(v') = h(v) \in T$ or (b) $h(v) \overset{a}{\Longrightarrow} h(v')$
4. $\forall v \in V. (v \notin T$ and $h(v) \notin T)$ implies $\mathcal{I}(v) = \mathcal{I}(h(v))$

Intuitively, nodes v in a ready tree represent stable states $h(v)$ of p (cf. Cond. (1), first part) and transitions represent computations containing exactly one observable action (cf. Cond. (3)(b)). Since computations do not contain *false*–states, no represented state is in F (cf. Cond. (1), second part). Since p might not be stable, the root v_0 of a ready tree represents a stable state reachable from p by some internal computation (cf. Cond. (2)). If the state $h(v)$ represented by node v is in T, the subtree of v is arbitrary since $h(v)$ is considered to be completely underspecified (cf. Conds. (3)(a) and (4)). In case $h(v) \notin T$, one distinguishes two cases: (i) if $v \notin T$, then v and $h(v)$ must have the same initial actions, i.e., the same *ready set*; (ii) if $v \in T$, the observation stops at this node and nothing is required in Conds. (3) and (4).

In the following, we write $\mathrm{RT}(p)$ for the set of all ready trees of p, $\mathrm{fRT}(p)$ for the set of all ready trees of p that have finite depth (*finite–depth* ready trees), and $\mathrm{cRT}(p)$ for the set of ready trees $\langle V, \longrightarrow, T, \emptyset\rangle$ where $T = \emptyset$ (*complete* ready trees). Note that a complete ready tree is called *complete* as it never stops its task of observing; hence, complete ready trees are often infinite in practice. Moreover, *false*–states may be characterised as follows:

Lemma 8. *$RT(p) = \emptyset$ if and only if $p \in F$.*

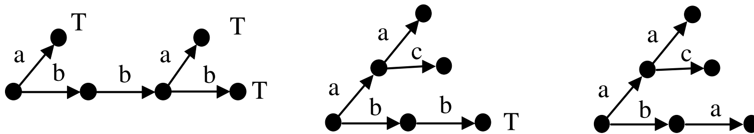


Fig. 5. Some ready trees of *spec*

We illustrate our concept of ready trees by returning to our example of Fig. 4. Some of the ready trees of process *spec* are shown in Fig. 5. In the first ready tree, the observation stops after the third b. In the second tree, we see that we can observe an arbitrary tree after a, since the respective state of *spec* is in T. An arbitrary tree can also consist of just the root, as shown for the right–most a in the third tree; this tree is also complete. Process *impl* in Fig. 4 has only one complete ready tree which is an infinite path repeating sequence abc; this is also a ready tree of *spec*.

Ready–tree preorder & expressiveness. Our ready–tree semantics suggests the following refinement preorder:

Definition 9 (Ready–Tree Preorder). The *ready–tree preorder* $\sqsubseteq$ on processes is defined as reverse ready–tree inclusion, i.e., $p \sqsubseteq q$ if $\mathrm{RT}(q) \subseteq \mathrm{RT}(p)$.

This preorder will turn out to be the desired fully–abstract preorder contained in the naive consistency preorder. We first show that $\sqsubseteq$ could just as well be formulated on the basis of complete ready trees and, for finitely branching LTS, of finite–depth ready trees. A crucial notion for our results is the following:

Definition 10 (Ready–Tree Prefix). Ready tree v_0 is *prefix* of ready tree w_0, in signs $v_0 \leq w_0$, if there exists an injective mapping $\rho : V \hookrightarrow W$ such that:

1. $\rho(v_0) = w_0$
2. $v \stackrel{a}{\longrightarrow} v' \implies \rho(v) \stackrel{a}{\longrightarrow} \rho(v')$
3. $\rho(v) \stackrel{a}{\longrightarrow} w' \implies v \in T$ or $(\exists v'.\, v \stackrel{a}{\longrightarrow} v'$ and $\rho(v') = w')$
4. $\rho(v) \in T \implies v \in T$

Intuitively, one observation is a prefix of another if it stops observing earlier. Recall that a *true*–node indicates that observation stops (cf. Cond. (3)). Intuitively, we obtain a prefix of w_0 by cutting all transitions from some nodes (and adding them to T), while cutting just some transitions of a node is not allowed.

Lemma 11. $\{v_0 \,|\, \exists w_0 \in cRT(p).\, v_0 \leq w_0\} = RT(p)$.

As a consequence, we obtain the following corollary:

Corollary 12

1. *$RT(p)$ is uniquely determined by $cRT(p)$, and vice versa.*
2. $RT(p) \subseteq RT(q) \iff cRT(p) \subseteq cRT(q)$
3. $fRT(p) = \{v_0 \text{ of finite depth} \,|\, \exists w_0 \in cRT(p).\, v_0 \leq w_0\}$

Before stating the next lemma we introduce the following definitions that allow us to approximate ready trees:

Definition 13 (k–Ready Tree). A *k–tree* $\langle V, \longrightarrow, T, \emptyset\rangle$, where $k \in \mathbb{N}_0$, is an observation tree where all nodes have depth at most k, and T is the set of all nodes of depth k. A *k–ready tree* of p is a ready tree of p that is also a k–tree. Moreover, $k\text{–RT}(p) =_{\text{df}} \{v_0 \in \text{RT}(p) \,|\, v_0 \text{ is a } k\text{–tree}\}$.

Intuitively, k–trees represent observations of k steps.

Definition 14 (Limit). Let $\boldsymbol{v}$ be an infinite sequence $(v_k)_{k\in\mathbb{N}}$ where $v_k \in k\text{–RT}(p)$ and $v_k \leq v_{k+1}$, with the identity as injection, for all $k \in \mathbb{N}$. Then, $\lim \boldsymbol{v}$ is the component–wise union of all v_k with T set to empty; $\lim \boldsymbol{v}$ is called a *limit* of p.

Observe that a node of some v_k in such a sequence is not in T_{k+1}, whence nodes in T are successively pushed out. In the limit, we may thus set T to empty. Moreover, if $v_k = v_{k+1} = v_{k+2} = \ldots$ for some k, then the limit is v_k; this happens exactly when v_k is complete. According to the following definition, we base the notion of finite branching on the weak transition relation $\stackrel{\alpha}{\Longrightarrow}$.

Definition 15 (Finite Branching). A process p is finite branching if, for every p' reachable from p, there are only finitely many $\langle \alpha, p''\rangle$ with $p' \stackrel{\alpha}{\Longrightarrow} p''$.

For finite–branching processes p, cRT(p) is characterised by the limits of p.

Lemma 16. *If p is finite branching, $cRT(p)$ equals the set of all limits of p.*

Note that the premise "p is finite branching" is only needed for direction "$\supseteq$" in the above lemma. We may now obtain the following corollary of Cor. 12(3) and

of Lemma 16, which is key to proving compositionality and full abstraction of our ready–tree preorder in the next section.

Corollary 17

1. $cRT(p) \subseteq cRT(q) \implies fRT(p) \subseteq fRT(q)$, *always.*
2. $cRT(p) \subseteq cRT(q) \impliedby fRT(p) \subseteq fRT(q)$, *if p is finite branching.*

4 Compositionality and Full Abstraction

This section establishes our full–abstraction result of the ready–tree preorder $\lesssim$ with respect to the consistency testing preorder $\sqsubseteq$, and proves that $\wedge$ and $\vee$ are indeed conjunction and, respectively, disjunction for $\lesssim$. To do so, we first show that $\wedge$ and $\vee$ correspond to intersection and union on the semantic level, respectively. While the correspondence for $\vee$ holds for ready trees in general, the correspondence for $\wedge$ only holds for *complete* ready trees.

Theorem 18 (Set–Theoretic Interpretation of $\wedge$ and $\vee$)
1. $cRT(p \wedge q) = cRT(p) \cap cRT(q)$ *2.* $RT(p \vee q) = RT(p) \cup RT(q)$

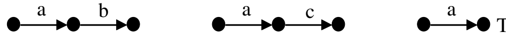


Fig. 6. Necessity of considering complete ready trees for conjunction

Fig. 6 illustrates that Thm. 18(1) is invalid when considering all ready trees instead of complete ready trees. The two processes displayed on the left and in the middle have the ready tree displayed on the right in common. However, the conjunction of the two processes is false and has no ready trees. Intuitively, the shown common ready tree formalises an observation that finished too early to encounter the inconsistency. Compositionality of our conjunction and disjunction operators for $\lesssim$ is now an immediate consequence of Thm. 18:

Theorem 19 (Compositionality)
1. $p \lesssim q \implies p \wedge r \lesssim q \wedge r$ *2.* $p \lesssim q \implies p \vee r \lesssim q \vee r$

Thm. 18 also allows us to prove that $\wedge$ and $\vee$ really behave as conjunction and disjunction with respect to our refinement relation.

Theorem 20 ($\wedge$ is And & $\vee$ is Or)
1. $p \wedge q \lesssim r \iff p \lesssim r$ *and* $q \lesssim r$ *2.* $r \lesssim p \vee q \iff r \lesssim p$ *and* $r \lesssim q$

In order to see that ready trees are indeed fully–abstract with respect to our naive consistency preorder, it now suffices to prove that $\lesssim$ coincides with our consistency testing preorder. This means that $\lesssim$ is *the* adequate preorder in our setting of logical LTSs with conjunction and disjunction.

Theorem 21 (Full Abstraction) $\sqsubseteq \, = \, \underset{\sim}{\sqsubset}$

The following proposition states the validity of several boolean properties desired of conjunction and disjunction operators. Here, $=$ denotes the kernel of our consistency testing preorder (ready–tree preorder).

Proposition 22 (Properties of $\wedge$ and $\vee$)

Commutativity:	$p \wedge q = q \wedge p$	$p \vee q = q \vee p$
Associativity:	$(p \wedge q) \wedge r = p \wedge (q \wedge r)$	$(p \vee q) \vee r = p \vee (q \vee r)$
Idempotence:	$p \wedge p = p$	$p \vee p = p$
False:	$p \wedge \mathit{ff} = \mathit{ff}$	$p \vee \mathit{ff} = p$
True:	$p \wedge \mathit{tt} = p$	$p \vee \mathit{tt} = \mathit{tt}$
Distributivity:	$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$

These properties follow directly from Thm. 18 and Cor. 12(2), as do the following:

Proposition 23 (Relating $\wedge$, $\vee$ to $\sqsubseteq$)

1. $p \wedge q = q \iff p \sqsubseteq q$ *2.* $p \vee q = p \iff p \sqsubseteq q$

We conclude this section by briefly returning to the illustrative processes *spec* and *impl* of Fig. 4. We have already remarked that the only complete ready tree of the latter is also one of the former. Hence, by Thm. 21, *impl* is indeed a refinement of *spec* according to our ready–tree preorder. Considering the conjunction of these processes, also shown in Fig. 4, it might be easier to see this using Prop. 23(1).

5 Related Work

Traditionally, process–algebraic and temporal–logic formalisms are not mixed but co–exist side by side. Indeed, the process–algebra school often uses synchronous composition and internal choice to model conjunction and disjunction, respectively. The compositionality of classic process–algebraic refinement preorders, such as failures semantics [4] and must–testing [12], enables component–based reasoning. However, inconsistencies in specifications are not captured so that, e.g., the conjunctive composition of a and b is identified with deadlock rather than ff. In contrast, the temporal–logic school distinguishes between deadlock and ff, but does not support component–based refinement.

Much research on mixing operational and logical styles of specification avoids dealing with inconsistencies by translating one style into the other. On the one hand, operational content may be translated into logic formulas, as is implicitly done in Lamport's TLA [10] or in the work of Graf and Sifakis [8]. In these approaches, logical implication serves as refinement relation. On the other hand, logical content may be translated into operational content. This is the case in automata–theoretic work, such as Kurshan's work on ω–automata [9], which includes synchronous and asynchronous composition operators and uses maximal trace inclusion as refinement relation. However, both logical implication and trace inclusion are insensitive to deadlock.

A seminal approach to compositional refinement relations in a mixed setting was proposed by Olderog in [13], where process–algebraic constructs are combined with trace formulas expressed in a predicate logic. In this approach, trace formulas can serve as processes, but not vice versa. Thus, freely mixing operational and logical styles is not supported and, in particular, conjunction cannot be applied to processes. For his setting, Olderog develops a denotational semantics that is a slight variation of standard failures semantics. Remarkably, an inconsistent formula is given a semantics that is not an element of the appropriate domain, as is stated on pp. 172–173 of [13].

Recently, a more general approach to combining process–algebraic and temporal–logic approaches was proposed in two papers by Cleaveland and Lüttgen [5, 6], which adopt a variant of De Nicola and Hennessy's must–testing preorder [12] as refinement preorder. However, Cleaveland and Lüttgen have not successfully solved the challenge of defining a semantics that is both deadlock–sensitive and compositional, and in which the conjunction operator and the refinement relation are compatible in the sense of Prop. 23(1). Our work solves this problem in the basic setting of logical LTS. Key for the solution is our new understanding of inconsistency, which is reflected by the fact that we consider processes a and $a + b$ as inconsistent, whereas they were treated as consistent in [6]. Observe that also in failure semantics and must–testing, a and $a + b$ are inconsistent in the sense that they do not have a common implementation.

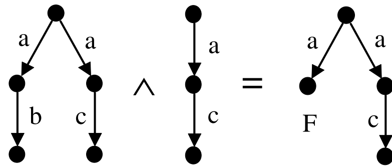


Fig. 7. Backward propagation of inconsistency

In addition, our backward propagation of inconsistency, as formalised in Def. 1(3), is in line with traditional semantics, as is illustrated in Fig. 7. The first conjunct would be a specification of the second conjunct with respect to failures semantics and must–testing, whence their conjunction should be consistent. In fact, the conjunction equals the second process in our ready–tree semantics.

Comparing Ready–Tree Semantics to Other Semantics

To the best of our knowledge, ready–tree semantics is novel and has not been studied in the literature before. We thus briefly compare it to three popular semantics, namely *ready–trace semantics*, *failures semantics* and *ready simulation* [7]. Since our treatment of divergence is different from the one of failures semantics, we restrict our discussion to τ–free processes.

A *ready trace* [1] of a process is a sequence of actions that it can perform and where, at the beginning of the trace, between any two actions and at the end, the ready set of the process reached at the respective stage is inserted. Such a

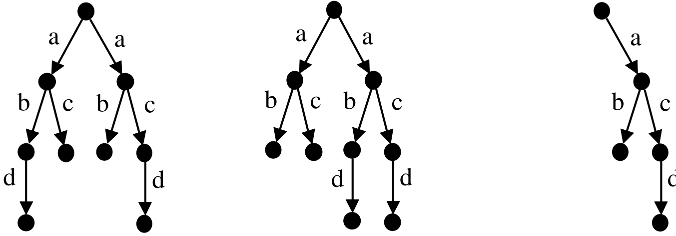


Fig. 8. Ready–tree semantics is strictly finer than ready–trace semantics

ready trace can be understood as a particular type of ready tree that consists only of a single path and includes additional transitions representing the ready sets. These additional transitions ensure that each state on the path has, for each action in its ready set, exactly one transition that either belongs to the path or ends in a *true*–state. For example, the first ready tree in Fig. 5 in Sec. 3 represents the ready trace $\{a, b\}b\{b\}b\{a, b\}$. Consequently, the ready traces of a process can be read off from its ready trees, and ready–tree inclusion implies ready–trace inclusion. The reverse implication does not hold: the two left–most processes in Fig. 8 possess the same ready traces; however, the observation tree on the right–hand side is a ready tree of the first, but not of the second process.

The *failures semantics* of a process is the set of its refusal pairs. Such a pair consists of a trace followed by a refusal set, i.e., a set of actions that the process reached by the trace cannot perform. Such a refusal pair can be read off from the respective ready trace by deleting all its ready sets and adding a set of actions having an empty intersection with the last ready set on the trace. Thus, ready–tree semantics is finer than failures semantics.

A process q *ready–simulates* some process p if there exists a simulation relation from p to q such that related states have identical ready sets. When tracing a ready tree of p, it is easy to see that such a simulation translates this ready tree to the same ready tree for q. Thus, the ready–tree preorder is coarser than ready simulation. Fig. 9 shows that it is indeed *strictly* coarser. Both processes displayed have the same ready trees; all of these trees are paths. However, the second process cannot even simulate the first process.

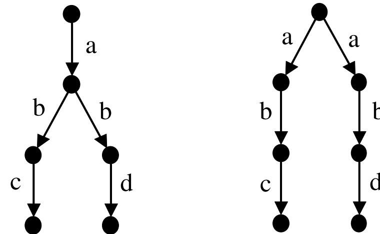


Fig. 9. The ready–tree preorder is strictly coarser than ready simulation

6 Conclusions and Future Work

This paper introduced a new semantics, the *ready–tree semantics*, that lies between ready–trace semantics and ready simulation. Our framework was one of τ–pure LTSs, with distinguished *true*– and *false*–states, and is equipped with

conjunction and *disjunction* operators. Key for defining the conjunction operator was the careful, inductive formalisation of an inconsistency predicate. The implied ready–tree preorder proved to be compositional and fully–abstract with respect to a naive preorder that allows inconsistent specifications to be refined only by inconsistent implementations. Standard laws of boolean algebra hold as expected, due to the fact that conjunction and disjunction on LTSs correspond to intersection and union on ready trees, respectively.

Consequently, this paper solves the problems of defining conjunction which are reported in closely related work [5, 6], albeit in a simpler setting that does not consider process–algebraic operators but only conjunction and disjunction. However, it is the simplicity of our setting that brought the subtleties of defining a fully–abstract semantics in the presence of conjunction to light, and which offered a way forward in addressing the challenge of defining "logical" process calculi, i.e., process calculi that allow one to freely mix process–algebraic and temporal–logic operators [6].

Future work shall extend our results to richer frameworks. Firstly, we plan to lift our requirement of τ–purity on LTS and extend our framework by standard process–algebraic operators such as parallel composition, hiding and recursion. In particular hiding is likely to prove challenging due to its transformation of observable infinite behaviour into divergent behaviour. Secondly, our framework shall be semantically extended from LTS to Büchi LTS [5] so that one may express liveness and fairness properties, and syntactically to linear–time temporal–logic formulas [6]. Last, but not least, we wish to explore tool support.

Acknowledgements. We thank Rance Cleaveland for many fruitful discussions and particularly for suggesting the use of an inconsistency predicate.

References

[1] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Ready-trace semantics for concrete process algebra with the priority operator. *Computer J.*, 30(6):498–506, 1987.
[2] J.A. Bergstra, A. Ponse, and S.A. Smolka. *Handbook of Process Algebra*. Elsevier Science, 2001.
[3] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995.
[4] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
[5] R. Cleaveland and G. Lüttgen. A semantic theory for heterogeneous system design. In *FSTTCS 2000*, vol. 1974 of *LNCS*, pp. 312–324. Springer-Verlag, 2000.
[6] R. Cleaveland and G. Lüttgen. A logical process calculus. In *EXPRESS 2002*, vol. 68,2 of *ENTCS*. Elsevier Science, 2002.
[7] R. van Glabbeek. The linear time – branching time spectrum II. In *CONCUR '93*, vol. 715 of *LNCS*, pp. 66–81. Springer-Verlag, 1993.
[8] S. Graf and J. Sifakis. A logic for the description of non-deterministic programs and their properties. *Information and Control*, 68(1–3):254–270, 1986.
[9] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton Univ. Press, 1994.

[10] L. Lamport. The temporal logic of actions. *TOPLAS*, 16(3):872–923, 1994.
[11] G. Lüttgen and W. Vogler. Conjunction on processes: Full-abstraction via ready-tree semantics. Tech. Rep. YCS-2005-396, Dept. of Comp. Sci., Univ. of York, UK, 2005.
[12] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1983.
[13] E.R. Olderog. *Nets, Terms and Formulas.* Cambridge Tracts in Theoretical Computer Science 23. Cambridge Univ. Press, 1991.
[14] A. Pnueli. The temporal logic of programs. In *FOCS '77*, pp. 46–57. IEEE Computer Society Press, 1977.

Undecidability Results for Bisimilarity on Prefix Rewrite Systems

Petr Jančar[1,⋆] and Jiří Srba[2,⋆⋆]

[1] Center of Applied Cybernetics, Department of Computer Science, Technical University of Ostrava, Czech Republic
[2] **BRICS**[⋆⋆⋆] Department of Computer Science, Aalborg University, Denmark

Abstract. We answer an open question related to bisimilarity checking on labelled transition systems generated by prefix rewrite rules on words. Stirling (1996, 1998) proved the decidability of bisimilarity for normed pushdown processes. This result was substantially extended by Sénizergues (1998, 2005) who showed the decidability for regular (or equational) graphs of finite out-degree (which include unnormed pushdown processes). The question of decidability of bisimilarity for a more general class of so called Type -1 systems (generated by prefix rewrite rules of the form $R \xrightarrow{a} w$ where R is a regular language) was left open; this was repeatedly indicated by both Stirling and Sénizergues. Here we answer the question negatively, i.e., we show undecidability of bisimilarity on Type -1 systems, even in the normed case. We complete the picture by considering classes of systems that use rewrite rules of the form $w \xrightarrow{a} R$ and $R_1 \xrightarrow{a} R_2$ and show when they yield low undecidability (Π^0_1-completeness) and when high undecidability (Σ^1_1-completeness), all with and without the assumption of normedness.

1 Introduction

Bisimilarity [17], or bisimulation equivalence, has been recognized as a fundamental notion in concurrency theory, in verification of behaviour of (reactive) systems, and in other areas. This has initiated several research directions, one of them exploring the decidability and complexity questions for bisimilarity. The obtained results differ from those known in the case of classical language equivalence; we can refer to surveys like [3, 25].

Bisimilarity is defined on labelled transition systems which can be viewed as (possibly infinite) edge-labelled directed graphs. Particular classes of graphs which have been in the focus of researchers are defined by prefix rewrite systems. We refer to a hierarchy defined by Stirling [27], which is interconnected with the work of Caucal [7, 5, 9]. We focus on (subclasses of) so called Type -2 systems;

⋆ The author is supported by the Czech Ministry of Education, Grant No. 1M0567.
⋆⋆ The author is supported in part by the research center ITI, project No. 1M0545.
⋆⋆⋆ **B**asic **R**esearch in **C**omputer **S**cience, Centre of the Danish National Research Foundation.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 277–291, 2006.
© Springer-Verlag Berlin Heidelberg 2006

such a system is given by a finite set of rewrite rules of the form $R_1 \xrightarrow{a} R_2$ where a is an action name (i.e. edge-label) and R_1, R_2 are regular languages over a finite alphabet. Processes (or states in the respective labelled transition system) are finite sequences of alphabet symbols. A rule $R_1 \xrightarrow{a} R_2$ stands for a potentially infinite set of rules $\{w \xrightarrow{a} w' \mid w \in R_1,\ w' \in R_2\}$, where $w \xrightarrow{a} w'$ can be applied to a process v iff w is a prefix of v (which is then replaced by w'). A process v is called *normed* iff each (finite) path from v can be prolonged to reach the empty process (word) ε.

Important subclasses of Type -2 graphs are called Type -1 and Type 0, where rules are of the form $R \xrightarrow{a} w$ and $w \xrightarrow{a} w'$, respectively. The class of Type 0 systems is isomorphic to the class of pushdown graphs [7], also called Type $1\frac{1}{2}$ by Stirling. By imposing further restrictions we get Type 2 graphs which correspond to BPA (Basic Process Algebra) and Type 3 graphs which coincide with finite-state transition systems.

Several nontrivial results achieved for pushdown (Type 0) processes turned out to be extendable to a superclass of Type -2 systems, namely the class of prefix-recognizable graphs, also called REC_{RAT} in [9]. This includes e.g. decidability of monadic second order logic [9] and the existence of uniform winning strategies for parity games [6]. The decidability questions for bisimilarity are, however, more intricate.

In 1995 Caucal [8] formulated three open questions about decidability of bisimilarity for (1) pushdown graphs, (2) regular graphs of finite out-degree, and (3) regular graphs. A bit later, Stirling showed the decidability of bisimilarity for restricted, namely normed, pushdown processes [27]. He stated the following two questions:

– Is bisimilarity decidable for Type -1 systems?
– Is bisimilarity decidable for Type -2 systems?

The initial hope was that the technique for normed pushdown processes might be extendable to these classes, in the normed case at least. Caucal's problem (1) was answered positively (in the full, i.e., unrestricted case) by Sénizergues [19] (who extended the technique used in his famous result for deterministic pushdown automata [20]). Stirling later presented a shorter proof in [29]. The result of Sénizergues also gives a positive answer to Caucal's problem (2); a complete journal version appeared recently in [21]. Due to a terminology mismatch, it was incorrectly indicated in [19] that the positive decidability result applies to Type -1 systems as well; this was later corrected in [21] by noting that it is valid (just) for a significant subclass of Type -1 graphs. More precisely, in [21] it was shown that regular graphs of finite out-degree (for which the decidability result was obtained) coincide up to isomorphism with collapsed graphs of pushdown automata with deterministic and popping only ε-transitions, and this is not the full class of Type -1 systems (where nondeterministic popping is allowed).

Remark. In the full paper (available as a BRICS technical report) we show that the class of regular graphs of finite out-degree can be characterized by Type -1 rules $R \xrightarrow{a} w$ with the restriction that R is a *prefix-free* (regular) language.

Thus Stirling's question about decidability of bisimilarity for Type -1 systems, also in the normed case, remained open (as several times explicitly indicated in the literature, most recently in [21]).

Our contribution. The main contribution of the present paper is the result showing the undecidability of bisimilarity on Type -1 systems, even in the normed case. Hence we have answered negatively the two open problems formulated by Stirling. Besides this, we have performed a more detailed analysis of the undecidability on related process classes.

We have also slightly extended the considered hierarchy. We view Stirling's Type -1 rules $R \xrightarrow{a} w$ as Type -1a, and we introduce a complementary class called Type -1b to denote rules of the type $w \xrightarrow{a} R$. Such a comparative study provides a deeper insight into prefix rewriting systems by classifying the respective undecidability degrees.

Remark. In the full paper we show that the classes -1a and -1b are incomparable w.r.t. bisimilarity and strictly above Type 0 and below Type -2 systems.

Let us recall a general experience that the 'natural' undecidable problems we encounter in computer science are either 'lowly' undecidable, i.e., at the first levels of arithmetical hierarchy — typically equivalent to the halting problem or its complement (Σ^0_1-complete or Π^0_1-complete), or 'highly' undecidable — typically complete for the first levels of analytical hierarchy (Σ^1_1-complete or Π^1_1-complete).

We demonstrate that bisimilarity is undecidable for normed Type -1a and Type -1b processes. More precisely, we establish Π^0_1-completeness of bisimilarity, both in the normed case and the unrestricted case, for Type -1a systems and in the normed case also for Type -1b systems. High undecidability, in fact Σ^1_1-completeness, is shown in the unrestricted case for Type -1b systems, and in the normed case and the unrestricted case for Type -2 systems. These results are completed by an observation that normedness of a given (Type -2) process is decidable.

Last but not least, our results are achieved in a uniform way, using reductions from two variants of Post's Correspondence Problem (one Π^0_1-complete, the other Σ^1_1-complete). An important technical ingredient of our reductions is the so called Defender's Choice technique (related to bisimulation games), which we most recently used in [14].

Remark. The techniques from the undecidability proofs of weak bisimilarity for pushdown automata [23, 24] can be used to prove undecidability (Σ^1_1-completeness) of strong bisimilarity also for Type -2 systems (no conflict between visible and ε-transitions means that ε-collapsed PDA graphs coincide with Type -2 graphs [29]; even though the pushdown rules in [23, 24] do not avoid the mentioned conflict, they can be modified to suppress clashes between visible and ε-moves). Nevertheless, the constructions from [23, 24] use pushdown processes the collapsed graphs of which have infinite out-degree and it is not straightforward to adapt the reductions to work also for Type -1a systems.

2 Preliminaries

A *labelled transition system* (LTS) is a triple $(S, \mathcal{A}ct, \longrightarrow)$ where S is a set of *states* (or *processes*), $\mathcal{A}ct$ is a set of *labels* (or *actions*), and $\longrightarrow \subseteq S \times \mathcal{A}ct \times S$ is a *transition relation*; for each $a \in \mathcal{A}ct$, we view $\stackrel{a}{\longrightarrow}$ as a binary relation on S where $\alpha \stackrel{a}{\longrightarrow} \beta$ iff $(\alpha, a, \beta) \in \longrightarrow$. The notation can be naturally extended to $\alpha \stackrel{s}{\longrightarrow} \beta$ for finite sequences of actions s; and by $\alpha \longrightarrow^* \beta$ we mean that there is s such that $\alpha \stackrel{s}{\longrightarrow} \beta$.

Given $(S, \mathcal{A}ct, \longrightarrow)$, a binary relation $R \subseteq S \times S$ is a *simulation* iff for each $(\alpha, \beta) \in R$, $a \in \mathcal{A}ct$, and α' such that $\alpha \stackrel{a}{\longrightarrow} \alpha'$ there is β' such that $\beta \stackrel{a}{\longrightarrow} \beta'$ and $(\alpha', \beta') \in R$. A *bisimulation* is a simulation which is symmetric. Processes α and β are *bisimilar*, denoted $\alpha \sim \beta$, if there is a bisimulation containing (α, β). We note that bisimilarity is an equivalence relation.

We shall use a standard game-theoretic characterization of bisimilarity [31, 26]. A *bisimulation game* on a pair of processes (α_1, α_2) is a two-player game between *Attacker* and *Defender*. The game is played in *rounds*. In each round (consisting of two moves) the players change the *current pair of states* (β_1, β_2) (initially $\beta_1 = \alpha_1$ and $\beta_2 = \alpha_2$) according to the following rule:

1. Attacker chooses $i \in \{1, 2\}$, $a \in \mathcal{A}ct$ and $\beta'_i \in S$ such that $\beta_i \stackrel{a}{\longrightarrow} \beta'_i$. He thus creates an *intermediate pair* which is (β'_1, β_2) in the case $i = 1$, and (β_1, β'_2) in the case $i = 2$.
2. Defender responds by choosing $\beta'_{3-i} \in S$ such that $\beta_{3-i} \stackrel{a}{\longrightarrow} \beta'_{3-i}$.
3. The pair (β'_1, β'_2) becomes the (new) current pair of states.

Any *play* (of the bisimulation game) thus corresponds to a sequence of pairs of states such that Attacker is making a move from every odd position and Defender from every even one (under the same action as in the previous Attacker's move).

A play (and the corresponding sequence) is finite iff one of the players gets stuck (cannot make a move); the player who got stuck lost the play and the other player is the winner. (A play finishing in an intermediate pair on an even position is winning for Attacker and a play finishing on an odd position is winning for Defender.) If the play is infinite then Defender is the winner. We use the following standard fact.

Proposition 1. *It holds that $\alpha_1 \sim \alpha_2$ iff Defender has a winning strategy in the bisimulation game starting with the pair (α_1, α_2), and $\alpha_1 \not\sim \alpha_2$ iff Attacker has a winning strategy.*

We shall now demonstrate a simple idea to establish semidecidability of nonbisimilarity for a particular class of LTSs; the idea slightly extends the standard argument used for image-finite systems [12]. (For example, for normed systems of Type -1b considered in the proof of Theorem 2, we are able to argue about the semidecidability of nonbisimilarity even though the systems are not necessarily image-finite.)

We say that a labelled transition system $(S, \mathcal{A}ct, \longrightarrow)$ is *effective* iff both S and $\mathcal{A}ct$ are decidable subsets of the set of all finite strings in a given finite alphabet and the relation $\longrightarrow$ is decidable.

An LTS $(S, \mathcal{A}ct, \longrightarrow)$ is called *finitely over-approximable* (w.r.t. bisimilarity) iff for any $\alpha, \beta \in S$, $a \in \mathcal{A}ct$, a finite set $E_{(\alpha,\beta,a)} \subseteq S$ can be effectively constructed such that whenever $\beta \xrightarrow{a} \beta'$ and $\beta' \sim \alpha$ then $\beta' \in E_{(\alpha,\beta,a)}$. Thus the (finite) set $E_{(\alpha,\beta,a)}$ over-approximates the set of a-successors of β which are bisimilar with α.

Proposition 2. *The problem of nonbisimilarity on effective and finitely over-approximable labelled transition systems is semidecidable, i.e., the bisimilarity problem is in* Π_1^0.

Proof. (Sketch) It is sufficient to provide a procedure which halts, for a given pair (α_1, α_2) of a given effective and finitely over-approximable system, iff there is a winning strategy for Attacker. Such a strategy can be naturally viewed as a tree where each vertex is labelled by a pair of processes and each edge, labelled by an action, corresponds to a move (of Attacker or Defender). While each vertex on an odd level has just one outgoing edge (Attacker's moves are fixed by the strategy), the vertices on even levels (corresponding to the 'intermediate' pairs) can have more successors (corresponding to Defender's choices). Each branch of the tree corresponds to a possible play when Attacker plays according to the assumed winning strategy; each branch is thus finite (finishing by Defender's getting stuck). Due to the assumed finite over-approximability, it is always sufficient to consider only finitely many possibilities for Defender's moves; the respective strategy-tree is then finitely branching and thus finite. So it is sufficient to systematically generate all finite trees and check for each of them if it happens to represent a winning strategy of Attacker; the checking can be done algorithmically due to our effectiveness assumptions. □

2.1 A Hierarchy of Regular Prefix Rewriting

We are interested in special labelled transition systems, namely those generated by systems of prefix rewrite rules. We now provide the relevant definitions.

The most general systems we consider are *Type -2 systems*. Such a system $\mathcal{S}$ can be viewed as a triple $\mathcal{S} = (\Gamma, \mathcal{A}ct, \Delta)$ where Γ is a finite set of *process symbols*, $\mathcal{A}ct$ is a finite set of *actions*, and Δ is a finite set of *rewrite rules*. Each rewrite rule is of the form $R_1 \xrightarrow{a} R_2$ where $a \in \mathcal{A}ct$ and R_1 and R_2 are regular languages over Γ such that $\varepsilon \notin R_1$; for concreteness, we can assume that R_1, R_2 are given by regular expressions.

We view the system $\mathcal{S}$ as generating a certain LTS $(\Gamma^*, \mathcal{A}ct, \longrightarrow)$. A process (or a state in the respective LTS) is any finite sequence of process symbols, i.e., any element of Γ^*; we shall use $u, v, w, \ldots$ for denoting elements of Γ^*, and ε for denoting the empty sequence. The transition relation $\longrightarrow$ (i.e., the collection of relations $\xrightarrow{a}$) is defined by the following derivation rule:

$$\frac{(R_1 \xrightarrow{a} R_2) \in \Delta, \quad w \in R_1, \quad w' \in R_2, \quad u \in \Gamma^*}{wu \xrightarrow{a} w'u}$$

Thus any rule $(R_1 \stackrel{a}{\longrightarrow} R_2) \in \Delta$ represents possibly infinitely many rewrite rules $w \stackrel{a}{\longrightarrow} w'$ where $w \in R_1$ and $w' \in R_2$.

We shall also need the notion of normedness. We say that a process $w \in \Gamma^*$ is *normed* if for any w' such that $w \longrightarrow^* w'$ we have $w' \longrightarrow^* \varepsilon$. In other words, a process w is normed iff any path from w in the respective LTS can be prolonged to finish in ε. A *norm* of a normed process w, denoted by $norm(w)$, is the length of the shortest action sequence s such that $w \stackrel{s}{\longrightarrow} \varepsilon$.

Proposition 3. *If two normed processes are bisimilar then they have the same norm.*

Proof. Assume normed u, v with $norm(u) < norm(v)$. For the shortest sequence s such that $u \stackrel{s}{\longrightarrow} \varepsilon$ we have: if $v \stackrel{s}{\longrightarrow} v'$ then v' is normed and $v' \neq \varepsilon$ (thus v' can perform an action). This implies that u and v are not bisimilar. □

Proposition 4. *There is an algorithm which given a Type -2 process v decides whether v is normed, and computes its norm in the positive case.*

Proof. (Sketch) We can base the algorithm on the well-known fact regarding (classical) pushdown automata: given a pushdown automaton and an initial state × stack configuration, the set of all state × stack configurations reachable from the initial one is regular, and its representation can be effectively constructed [2, 10].

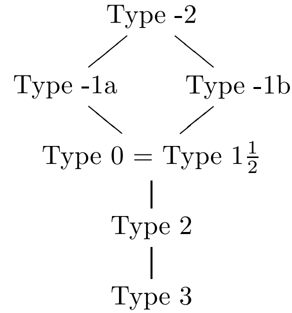
We observe (see also [29]) that applying a rule $R_1 \stackrel{a}{\longrightarrow} R_2$ to v, i.e., replacing a prefix $w \in R_1$ of v by $w' \in R_2$, can be implemented by a series of ε-moves of a pushdown automaton (whose control unit includes finite automata for R_1, R_2).

In this way we can easily derive that, given a Type -2 system and a process v, the set $post^*(v)$ — consisting of all processes reachable from v — is an effectively constructible regular set. Similarly, the set $pre^*(\varepsilon)$ — consisting of all processes from which ε is reachable — is an effectively constructible regular set. Checking normedness of v now amounts to verifying whether $post^*(v) \subseteq pre^*(\varepsilon)$.

Computing $norm(v)$ for a normed v can be accomplished by stepwise constructing $pre(\varepsilon)$, $pre(\,pre(\varepsilon)\,)$, $pre(\,pre(\,pre(\varepsilon)\,)\,)$, ... until v is included; here $pre(R)$ denotes the set of processes from which some $u \in R$ is reachable in one step. Such a computation can be again easily reduced to computing the sets of reachable configurations of pushdown automata. □

The other systems we consider arise from the above defined Type -2 systems by restricting the form of rewrite rules. We use the terminology introduced by Stirling (see, e.g., [30]). In the following table, R_1, R_2 and R stand for regular sets over Γ; w, w' stand for elements of Γ^* (the respective regular languages are thus singletons); and X, Y, p, q stand for elements of Γ. We have added Type -1b to Stirling's table; his Type -1 coincides with our Type -1a. In the full paper it is shown that Type -1a and -1b classes are incomparable w.r.t. bisimilarity and strictly above Type 0 and below Type -2 systems.

Type	Form of Rewrite Rules
Type -2	$R_1 \stackrel{a}{\longrightarrow} R_2$
Type -1a/-1b	$R \stackrel{a}{\longrightarrow} w$ / $w \stackrel{a}{\longrightarrow} R$
Type 0	$w \stackrel{a}{\longrightarrow} w'$
Type $1\frac{1}{2}$	$pX \stackrel{a}{\longrightarrow} qw$
Type 2	$X \stackrel{a}{\longrightarrow} w$
Type 3	$X \stackrel{a}{\longrightarrow} Y,\ X \stackrel{a}{\longrightarrow} \varepsilon$



We can note that Type $1\frac{1}{2}$ rules are classical pushdown rules (p, q are 'highlighted' as finite control states and are disjoint with the stack alphabet); this class was shown to coincide up to isomorphism with Type 0 systems [7]. Type 2 systems are also called BPA (Basic Process Algebra) systems, and Type 3 systems correspond to finite labelled transition systems.

2.2 Versions of Post's Correspondence Problem

Here we recall the versions of Post's Correspondence Problem (PCP) which will be used in the later reductions.

A *PCP-instance* is a nonempty sequence $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ of pairs of nonempty words in the alphabet $\{A, B\}$ such that $|u_i| \leq |v_i|$ for all i, $1 \leq i \leq n$ (where $|u|$ denotes the length of u).

An *infinite initial solution* of the given instance is an infinite sequence of indices $i_1, i_2, i_3, \ldots$ from the set $\{1, 2, \ldots, n\}$ such that $i_1{=}1$ and the infinite words $u_{i_1} u_{i_2} u_{i_3} \cdots$ and $v_{i_1} v_{i_2} v_{i_3} \cdots$ are equal. An *infinite recurrent solution* is an infinite initial solution in which the index 1 appears infinitely often.

By *inf-PCP* we denote the problem to decide whether a given PCP instance has an infinite initial solution; *rec-PCP* denotes the problem to decide whether a given PCP instance has an infinite recurrent solution.

Proposition 5. *Problems inf-PCP and rec-PCP are Π^0_1-complete and Σ^1_1-complete, respectively.*

These facts can be easily established from well-known results but we can refer, e.g., to [18] for the (low) undecidability and to [11] for the high undecidability. Our (additional) requirement $|u_i| \leq |v_i|$ is non-standard but it can be easily checked to be harmless; we use it for its technical convenience. (The harmlessness of the extra requirement follows directly from the standard textbook reduction from Turing machines to PCP. The reduction produces an instance of PCP which satisfies our requirement, except for the last category of pairs of strings that are used to equalize the lengths of the two generated words in case that an accepting configuration is reached. As our question is about the existence of an infinite computation, we can safely omit the pairs from the last category.)

It is also useful to note the following obvious fact.

Proposition 6. *Given a PCP-instance* $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ *and a sequence* $i_1, i_2, i_3, \ldots$ *of indices where* $i_1 = 1$*, the following three conditions are equivalent:*

- $i_1, i_2, i_3, \ldots$ *is an infinite initial solution,*
- *for each* $\ell = 1, 2, 3, \ldots$*, the word* $u_{i_1} u_{i_2} \ldots u_{i_\ell}$ *is a prefix of* $v_{i_1} v_{i_2} \ldots v_{i_\ell}$*,*
- *for infinitely many* $\ell \geq 1$*, the word* $u_{i_1} u_{i_2} \ldots u_{i_\ell}$ *is a prefix of* $v_{i_1} v_{i_2} \ldots v_{i_\ell}$*.*

3 Proof Strategy

A crucial point in proving completeness results for bisimilarity on prefix rewriting systems are the hardness reductions, from inf-PCP or rec-PCP to the respective bisimilarity problems. Here we describe the general idea of these reductions, which will be implemented later by suitable sets of rewrite rules.

In each particular case studied in this paper, we assume a fixed PCP-instance $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ (over the alphabet $\{A, B\}$) and show how to construct an appropriate rewrite system $(\Gamma, \mathcal{A}ct, \Delta)$. The set of process symbols Γ will always contain the alphabet symbols A, B, 'index-symbols' $I_1, I_2, \ldots, I_n$, and auxiliary symbols $X, X', \bot$ and others.

Our constructions will guarantee that $XI_1\bot \sim X'I_1\bot$ (and hence that Defender has a winning strategy from the pair $(XI_1\bot\,,\ X'I_1\bot)$) if and only if there is an infinite initial solution or an infinite recurrent solution — depending on the source problem (inf-PCP or rec-PCP).

Our intention is that each play starting from $(\,XI_1\bot\,,\ X'I_1\bot\,)$ will begin with a *generating phase*: this phase produces pairs (of current states) of the form

$$(\,XI_{i_\ell}I_{i_{\ell-1}} \ldots I_{i_1}\bot\,,\ X'I_{i_\ell}I_{i_{\ell-1}} \ldots I_{i_1}\bot\,) \quad \text{(where } i_1{=}1\text{)} \tag{1}$$

where the players are stepwise building longer and longer prefixes of an infinite sequence $i_1, i_2, i_3, \ldots$; this means that the pair (1) can only be 'prolonged', i.e., followed by the pair

$$(XI_{i_m}I_{i_{m-1}} \ldots I_{i_{\ell+1}}I_{i_\ell}I_{i_{\ell-1}} \ldots I_{i_1}\bot\,,\ X'I_{i_m}I_{i_{m-1}} \ldots I_{i_{\ell+1}}I_{i_\ell}I_{i_{\ell-1}} \ldots I_{i_1}\bot)$$

for $m > \ell$. Moreover, in the case of rec-PCP we will guarantee that $i_m = 1$, which ensures that the first index has to be repeatedly inserted into the states.

Remark. Due to the nature of prefix rewrite rules, we represent generating of a sequence $I_{i_1}, I_{i_2}, I_{i_3}, \ldots$ by using the 'right-to-left' direction.

A subtle point is that the elements of the sequence $i_1, i_2, i_3, \ldots$ arising during the generating phase must be freely chosen by Defender. We implement this by a variant of so-called Defender's Choice technique (which we used, e.g., in [14]).

The generating phase can go on arbitrarily long, maybe forever (in which case Defender wins). Attacker will always have the possibility to finish this phase by *switching* to a *verification phase*; our rules will guarantee that Attacker can thus force his win from the current pair (1) if and only if $u_{i_1} u_{i_2} \ldots u_{i_\ell}$ is not a prefix of $v_{i_1} v_{i_2} \ldots v_{i_\ell}$.

The correctness of the described general strategy follows from Proposition 6.

4 (Low) Undecidability Results

In this section we show that bisimilarity is Π^0_1-complete for both normed and unrestricted Type -1a systems, and for normed Type -1b systems; this in particular entails the undecidability for (normed) Type -1a systems (i.e., Stirling's Type -1 systems).

As explained in Section 3, in what follows we assume a fixed PCP-instance $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ (over the alphabet $\{A, B\}$); the instance is now viewed as an input of inf-PCP.

4.1 Π^0_1-Completeness of Normed and Unrestricted Type-1a Systems

We first provide the rules and then explain how they implement the above described strategy. For the generating phase we add auxiliary process symbols Y and $Y_1, Y_2, \ldots, Y_n$, actions $1, 2, \ldots, n$ and c, and

(G1) rules:
$$X \xrightarrow{c} Y$$
$$X \xrightarrow{c} Y_i \qquad X' \xrightarrow{c} Y_i \qquad \text{for all } i \in \{1, 2, \ldots, n\}$$
$$Y \xrightarrow{i} XI_i \qquad Y_i \xrightarrow{i} X'I_i \qquad \text{for all } i \in \{1, 2, \ldots, n\}$$
$$Y_i \xrightarrow{j} XI_j \qquad \text{for all } i, j \in \{1, 2, \ldots, n\},\ i \neq j.$$

For switching we add the symbols C, C', an action d, and

(S1) rules:
$$X \xrightarrow{d} C$$
$$X(I^*)I_i \xrightarrow{d} C'w \qquad X'(I^*)I_i \xrightarrow{d} C'w \qquad \text{for all } i \in \{1, 2, \ldots, n\} \text{ and all suffixes } w \text{ of } v_i^R.$$

Notation. I^* stands for the regular expression $(I_1 + I_2 + \cdots + I_n)^*$; this uses the possibility allowed by Type -1a rules. For a word u, by u^R we denote the reverse image of u.

For the verification phase we add actions a, b, e, and the following rules in which we use a further piece of notation.

Notation. By $head(w)$ we denote the first symbol of w; $tail(w)$ is the rest of w. By $h(w)$ (head-action) we mean a when $head(w) = A$, and b when $head(w) = B$.

(V1) rules:
$$CA \xrightarrow{a} C \qquad C'A \xrightarrow{a} C'$$
$$CB \xrightarrow{b} C \qquad C'B \xrightarrow{b} C'$$
$$C\bot \xrightarrow{e} \varepsilon \qquad C'\bot \xrightarrow{e} \varepsilon$$
$$CI_i \xrightarrow{h(u_i^R)} C\ tail(u_i^R) \qquad C'I_i \xrightarrow{h(v_i^R)} C'\ tail(v_i^R)$$
$$\text{for all } i \in \{1, 2, \ldots, n\}$$

Let us now consider the system with the rules (G1), (S1), (V1), and the pair $(XI_1\bot,\ X'I_1\bot)$. Attacker can decide to perform a step of the generating phase by choosing the action c. He then has to use the rule $X \xrightarrow{c} Y$; any other c-move could be followed by Defender's response reaching syntactically equal processes

$(Y_iI_1\bot,\ Y_iI_1\bot)$ (which are trivially bisimilar). So it is Defender who (after Attacker's move $X \stackrel{c}{\longrightarrow} Y$) chooses some $i \in \{1,2,\ldots,n\}$ by performing the rule $X' \stackrel{c}{\longrightarrow} Y_i$. We thus get $(YI_1\bot,\ Y_iI_1\bot)$. The next action will be some $j \in \{1,2,\ldots,n\}$. If Attacker chooses $j \neq i$ then Defender installs syntactic equality $(XI_jI_1\bot,\ XI_jI_1\bot)$; so Attacker is forced to use the action i which means that the current round finishes in $(XI_iI_1\bot,\ X'I_iI_1\bot)$.

Attacker can decide to prolong the generating phase arbitrarily long but he always has the possibility to switch, by choosing the action d. In such case, from a current pair $(XI_{i_\ell}I_{i_{\ell-1}}\ldots I_{i_1}\bot,\ X'I_{i_\ell}I_{i_{\ell-1}}\ldots I_{i_1}\bot)$ he is forced to perform $X \stackrel{d}{\longrightarrow} C$ to avoid syntactic equality. So the 'left-hand side' process becomes $CI_{i_\ell}I_{i_{\ell-1}}\ldots I_{i_1}\bot$, and Defender installs some $C'wI_{i_m}I_{i_{m-1}}\ldots I_{i_1}\bot$ on the 'right-hand side', where $m < \ell$ and w is a suffix of $v^R_{i_{m+1}}$.

One can easily check that the rules (V1) guarantee

$$CI_{i_\ell}I_{i_{\ell-1}}\ldots I_{i_1}\bot \sim C'wI_{i_m}I_{i_{m-1}}\ldots I_{i_1}\bot \ \text{ iff } \ u_{i_1}\ldots u_{i_\ell} = v_{i_1}\ldots v_{i_m}w^R$$

and that Defender has the possibility to install such a bisimilar pair (by using the rule $X'(I^*)I_i \stackrel{d}{\longrightarrow} C'w$) iff $u_{i_1}\ldots u_{i_\ell}$ is a prefix of $v_{i_1}\ldots v_{i_\ell}$.

We also observe that $XI_1\bot$ and $X'I_1\bot$ are normed; we have thus proved the following lemma.

Lemma 1. *Problem inf-PCP is reducible to bisimilarity on normed Type -1a systems.*

Theorem 1. *Bisimilarity on Type -1a systems is Π^0_1-complete in both the normed case and the unrestricted case.*

Proof. Lemma 1 shows that bisimilarity is Π^0_1-hard already for normed Type -1a systems. Since (unrestricted) Type -1a systems are effectively image-finite, i.e., for each process w and every action a the set of a-successors of w is finite and effectively constructible, semidecidability of nonbisimilarity follows from Proposition 2. □

4.2 Π^0_1-Completeness of Normed Type -1b Systems

Normed Type -1b systems are handled very similarly as Type -1a, we only use different switching rules.

(S2) rules:

$$X' \stackrel{d}{\longrightarrow} C'$$

$$X \stackrel{d}{\longrightarrow} C(A+B)^* \qquad X' \stackrel{d}{\longrightarrow} C(A+B)^*$$

When now, i.e., in the system (G1), (S2), (V1), Attacker decides to switch to the verification phase, in a current pair $(XI_{i_\ell}I_{i_{\ell-1}}\ldots I_{i_1}\bot,\ X'I_{i_\ell}I_{i_{\ell-1}}\ldots I_{i_1}\bot)$, he is forced to use $X' \stackrel{d}{\longrightarrow} C'$ (on the right-hand side); Defender responds by extending the left-hand side. It is clear that there is an extension which guarantees Defender's win if and only if $u_{i_1}u_{i_2}\ldots u_{i_\ell}$ is a prefix of $v_{i_1}v_{i_2}\ldots v_{i_\ell}$.

Since $XI_1\bot$ and $X'I_1\bot$ are normed also in the system (G1), (S2), (V1), we have shown the following lemma.

Lemma 2. *Problem inf-PCP is reducible to bisimilarity on normed Type -1b systems.*

Theorem 2. *Bisimilarity on normed Type -1b systems is Π^0_1-complete.*

Proof. Π^0_1-hardness follows from Lemma 2. Type -1b systems are obviously effective, and for semidecidability of nonbisimilarity it is thus sufficient to show that *normed* Type -1b systems are finitely over-approximable and then use Proposition 2. (Note that Type -1b systems are in general not image-finite and hence the standard argument about semidecidability of the negative case does not directly apply.)

We recall that normed bisimilar processes must have equal norms (Proposition 3), and we note that $norm(u) \geq |u|/k$ where k is the length of the longest left-hand side in the rules $w \xrightarrow{a} R$ of the respective Type -1b system. Since $norm(u)$ is computable by Proposition 4, the required (finite) set $E_{(u,v,a)}$ for given processes u, v and an action a can be defined as $\{\, v' \mid |v'| \leq k \cdot norm(u) \,\}$. □

5 High Undecidability Results

We first note that (unrestricted) Type -2 systems represent a class of LTSs which satisfies the (straightforward) general criteria guaranteeing that the bisimilarity problem is in Σ^1_1 (see, e.g., [14]). (Processes u and v are bisimilar iff *there exists a set* of pairs which contains (u, v) and satisfies the conditions required by the definition of bisimulation.) So the main point in the following completeness results is to show Σ^1_1-hardness.

We again assume a fixed PCP-instance $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ (over the alphabet $\{A, B\}$); the instance is now viewed as an input of rec-PCP.

5.1 Σ^1_1-Completeness of (Unrestricted) Type -1b Systems

We modify the previously defined (normed) Type -1b system (G1), (S2), (V1). We first replace the (generating) rules (G1) with the following variant (G2), which repeatedly forces Defender to include the index 1 into the generated sequence. As before, I^* denotes $(I_1 + I_2 + \cdots + I_n)^*$.

(G2) rules:

$$X \xrightarrow{c} Y$$
$$X \xrightarrow{c} Y' I_1 I^* \qquad X' \xrightarrow{c} Y' I_1 I^*$$
$$Y' \xrightarrow{c} X'$$
$$Y \xrightarrow{c} X I^* \bot \qquad Y' \xrightarrow{c} X I^* \bot$$

Given a pair $(\, X I_{i_\ell} I_{i_{\ell-1}} \ldots I_{i_1} \bot ,\ X' I_{i_\ell} I_{i_{\ell-1}} \ldots I_{i_1} \bot \,)$ (with $i_1{=}i_\ell{=}1$), if Attacker decides to continue the generating phase (i.e., chooses the action c) then he is forced to perform $X \xrightarrow{c} Y$ (on the left-hand side), otherwise he loses. Defender responds by the rule $X' \xrightarrow{c} Y' I_1 I^*$ (on the right-hand side), i.e., he prolongs the right-hand side sequence by an arbitrarily chosen

finite segment finishing with I_1 (viewed from right to left). So we get the pair $(\,Y I_{i_\ell} I_{i_{\ell-1}} \dots I_{i_1} \bot\,,\; Y' I_{i_m} I_{i_{m-1}} \dots I_{i_{\ell+1}} I_{i_\ell} I_{i_{\ell-1}} \dots I_{i_1} \bot\,)$ where $m > \ell$ and i_m=1.

Now unnormedness comes 'into play'. Our rules maintain the property that the suffix after (the leftmost) $\bot$ does not matter. So Attacker is forced to perform $Y' \stackrel{c}{\longrightarrow} X'$ (on the right-hand side). Defender responds by 'killing' the left-hand side sequence (using a new occurrence of $\bot$) and creating a completely new one; important is that he has the possibility to install the pair

$$(\,X I_{i_m} I_{i_{m-1}} \dots I_{i_1} \bot w\,,\; X' I_{i_m} I_{i_{m-1}} \dots I_{i_1} \bot\,)$$

where w is unimportant and can be deemed omitted (which leaves the process in the same bisimilarity class).

However, we are not done yet. Unlike (G1), the new rules (G2) allow Defender to install an index sequence on the left-hand side which differs from the sequence he previously installed on the right-hand side. To prevent Defender from such 'cheating', we add some further switching and verification rules to (S2) and (V1). For this purpose we add new process symbols Z, Z' and an action f.

(S2') rules: (S2) and $\quad X \stackrel{f}{\longrightarrow} Z \qquad X' \stackrel{f}{\longrightarrow} Z'$

(V1') rules: (V1) and $\quad Z I_i \stackrel{i}{\longrightarrow} Z \qquad Z' I_i \stackrel{i}{\longrightarrow} Z' \qquad$ for all $i \in \{1, 2, \dots, n\}$

$$Z\bot \stackrel{e}{\longrightarrow} \varepsilon \qquad Z'\bot \stackrel{e}{\longrightarrow} \varepsilon$$

We remind the reader of the fact that even though the last two rules above remove the symbol $\bot$ from the sequence of process symbols, whatever remains after $\bot$ can only start with some process symbol from the set $\{I_1, \dots, I_n\}$ and hence the remaining process is stuck (no left-hand side of any rule in our system begins with any I_i). Type -1b system (G2), (S2'), (V1') thus proves the next lemma, from which the following theorem is derived.

Lemma 3. *Problem rec-PCP is reducible to bisimilarity on (unrestricted) Type -1b systems.*

Theorem 3. *Bisimilarity on (unrestricted) Type -1b systems is Σ_1^1-complete.*

5.2 Σ_1^1-Completeness of Normed and Unrestricted Type -2 Systems

Σ_1^1-completeness for (unrestricted) Type -2 systems follows immediately from the previous results (Type -1b is a subclass of Type -2). So we just have to show that normedness does not make the problem easier in this case.

We recall the unnormed Type -1b system (G2), (S2'), (V1'). It is sufficient to replace the last two rules of (G2) (resp. their left-hand sides); we thus get

(G3) rules:

$$\begin{array}{ll} X \stackrel{c}{\longrightarrow} Y & \\ X \stackrel{c}{\longrightarrow} Y' I_1 I^* & X' \stackrel{c}{\longrightarrow} Y' I_1 I^* \\ & Y' \stackrel{c}{\longrightarrow} X' \\ Y I^* \bot \stackrel{c}{\longrightarrow} X I^* \bot & Y' I^* \bot \stackrel{c}{\longrightarrow} X I^* \bot \quad . \end{array}$$

The processes $XI_1\bot$ and $X'I_1\bot$ in the resulting Type -2 system (G3), (S2'), (V1') are obviously normed (in any reachable process, $\bot$ can only occur as the last element in the sequence), and the correctness arguments remain the same. We have thus shown the following theorem.

Theorem 4. *Bisimilarity on Type -2 systems is Σ^1_1-complete in both the normed case and the unrestricted case.*

6 Conclusion and Final Remarks

We have answered negatively the open problem stated in 1996 by Stirling [27]: "Is strong bisimilarity decidable for Type -1 and Type -2 transition graphs?". A precise borderline between decidability and undecidability has been found: for Type -1a systems with rules of the form $R \stackrel{a}{\longrightarrow} w$ where R is a prefix-free regular language bisimilarity is decidable [21], while it is undecidable for the same class without the prefix-freeness restriction. We have also given a full characterization of the undecidability degrees of the studied problems. A summary of the results for bisimilarity checking is provided in the following table. Results without references were obtained in this paper.

	Normed Processes	**Unnormed Processes**
Type -2	Σ^1_1-complete	Σ^1_1-complete
Type -1b	Π^0_1-complete	Σ^1_1-complete
Type -1a	Π^0_1-complete	Π^0_1-complete
Type 0, and Type $1\frac{1}{2}$	decidable [28] EXPTIME-hard [16]	decidable [19, 21] EXPTIME-hard [16]
Type 2	$\in$ P [13] P-hard [1]	$\in$ 2-EXPTIME [4] PSPACE-hard [22]
Type 3	P-complete [15, 1]	P-complete [15, 1]

We note that the results for Type -1b systems illustrate a significant difference between normed and unnormed processes. An open problem is the precise complexity for PDA and BPA, and decidability of bisimilarity for unrestricted regular (equational) graphs. As a side result, our paper provides an alternative and easily understandable proof of undecidability of weak bisimilarity for normed pushdown processes since the class of ε-collapsed pushdown graphs is a superclass of Type -2 systems [29] and hence (high) undecidability of strong bisimilarity for normed Type -2 graphs implies (high) undecidability of weak bisimilarity for normed pushdown processes.

Acknowledgments. The authors thank to Géraud Sénizergues for a motivating discussion (at University of Stuttgart), to the anonymous referees for their useful remarks, and acknowledge a support from the Alexander von Humboldt Foundation.

References

1. J. Balcazar, J. Gabarro, and M. Santha. Deciding bisimilarity is P-complete. *Formal Aspects of Computing*, 4(6A):638–648, 1992.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. of the 8th International Conference on Concurrency Theory (CONCUR'97)*, vol. 1243 of *LNCS*, pages 135–150. Springer-Verlag, 1997.
3. O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, chapter 9, pages 545–623. Elsevier Science, 2001.
4. O. Burkart, D. Caucal, and B. Steffen. An elementary decision procedure for arbitrary context-free processes. In *Proc. of the 20th International Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, vol. 969 of *LNCS*, pages 423–433. Springer-Verlag, 1995.
5. O. Burkart, D. Caucal, and B. Steffen. Bisimulation collapse and the process taxonomy. In *Proc. of the 7th International Conference on Concurrency Theory (CONCUR'96)*, vol. 1119 of *LNCS*, pages 247–262. Springer-Verlag, 1996.
6. T. Cachat. Uniform solution of parity games on prefix-recognizable graphs. *Electronic Notes in Theoretical Computer Science*, 68(6), 2002.
7. D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, 1992.
8. D. Caucal. Bisimulation of context-free grammars and of pushdown automata. In *Modal Logic and Process Algebra*, vol. 53 of *CSLI Lectures Notes*, pages 85–106. University of Chicago Press, 1995.
9. D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. of the 23th International Colloquium on Automata, Languages and Programming (ICALP'96)*, vol. 1099, pages 194–205. Springer-Verlag, 1996.
10. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. of the 12th International Conference on Computer Aided Verification (CAV'00)*, vol. 1855 of *LNCS*, pages 232–247. Springer-Verlag, 2000.
11. D. Harel. Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness. *Journal of the ACM (JACM)*, 33(1): 224–248, 1986.
12. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, 1985.
13. Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158 (1–2):143–159, 1996.
14. P. Jančar and J. Srba. Highly undecidable questions for process algebras. In *Proc. of the 3rd IFIP International Conference on Theoretical Computer Science (TCS'04)*, Exploring New Frontiers of Theoretical Informatics, pages 507–520. Kluwer Academic Publishers, 2004.
15. P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.
16. A. Kučera and R. Mayr. On the complexity of semantic equivalences for pushdown automata and BPA. In *Proc. of the 27th International Symposium on Mathematical Foundations of Computer Science (MFCS'02)*, vol. 2420 of *LNCS*, pages 433–445. Springer-Verlag, 2002.

17. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
18. K. Ruohonen. Reversible machines and post's correspondence problem for biprefix morphisms. *Elektronische Informationsverarbeitung und Kybernetik*, 21(12):579–595, 1985.
19. G. Sénizergues. Decidability of bisimulation equivalence for equational graphs of finite out-degree. In *Proc. of the 39th Annual Symposium on Foundations of Computer Science(FOCS'98)*, pages 120–129. IEEE Computer Society, 1998.
20. G. Sénizergues. L(A)=L(B)? Decidability results from complete formal systems. *Theoretical Computer Science*, 251(1–2):1–166, 2001.
21. G. Senizergues. The bisimulation problem for equational graphs of finite out-degree. *SIAM Journal on Computing*, 34(5):1025–1106, 2005.
22. J. Srba. Strong bisimilarity and regularity of basic process algebra is PSPACE-hard. In *Proc. of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02)*, vol. 2380 of *LNCS*, pages 716–727. Springer-Verlag, 2002.
23. J. Srba. Undecidability of weak bisimilarity for pushdown processes. In *Proc. of the 13th International Conference on Concurrency Theory (CONCUR'02)*, vol. 2421 of *LNCS*, pages 579–593. Springer-Verlag, 2002.
24. J. Srba. Completeness results for undecidable bisimilarity problems. In *Proc. of the 5th International Workshop on Verification of Infinite-State Systems (INFINITY'03)*, vol. 98 of *ENTCS*, pages 5–19. Elsevier Science Publishers, 2004.
25. J. Srba. *Roadmap of Infinite results*, vol. 2: Formal Models and Semantics. World Scientific Publishing Co., 2004. Updated version can be downloaded from the author's home-page.
26. C. Stirling. Local model checking games. In *Proc. of the 6th International Conference on Concurrency Theory (CONCUR'95)*, vol. 962 of *LNCS*, pages 1–11. Springer-Verlag, 1995.
27. C. Stirling. Decidability of bisimulation equivalence for normed pushdown processes. In *Proc. of the 7th International Conference on Concurrency Theory (CONCUR'96)*, vol. 1119 of *LNCS*, pages 217–232. Springer-Verlag, 1996.
28. C. Stirling. Decidability of bisimulation equivalence for normed pushdown processes. *Theoretical Computer Science*, 195(2):113–131, 1998.
29. C. Stirling. Decidability of bisimulation equivalence for pushdown processes. Research Report EDI-INF-RR-0005, School of Informatics, Edinburgh University, January 2000. The latest version is downloadable from the author's home-page.
30. C. Stirling. Bisimulation and language equivalence. In *Logic for Concurrency and Synchronisation*, vol. 18 of *Trends in Logic*, pages 269–284. Kluwer Academic Publishers, 2003.
31. W. Thomas. On the Ehrenfeucht-Fraïssé game in theoretical computer science (extended abstract). In *Proc. of the 4th International Joint Conference CAAP/FASE, Theory and Practice of Software Development (TAPSOFT'93)*, vol. 668 of *LNCS*, pages 559–568. Springer-Verlag, 1993.

Propositional Dynamic Logic with Recursive Programs

Christof Löding[1] and Olivier Serre[2,⋆]

[1] RWTH Aachen, Germany
[2] LIAFA, Université Paris VII & CNRS, France

Abstract. We extend the propositional dynamic logic PDL of Fischer and Ladner with a restricted kind of recursive programs using the formalism of visibly pushdown automata (Alur, Madhusudan 2004). We show that the satisfiability problem for this extension remains decidable, generalising known decidability results for extensions of PDL by non-regular programs.

1 Introduction

Propositional Dynamic Logic (PDL) is a modal logic that was introduced by Fischer and Ladner in [5] to capture the behaviour of programs. The models for PDL formulas are transition systems whose edges are labelled with atomic programs and whose states are labelled with atomic propositions. Formulas and programs are inductively (and mutually) defined from atomic propositions and programs. Formulas are closed by the standard Boolean operations, and for each program α and each formula φ, $\langle\alpha\rangle\varphi$ is a formula meaning that there is an execution of program α that ends in a state where φ holds. A program is a regular language (represented by a regular expression of a finite automaton) over the set of atomic programs and tests (where tests correspond to formulas). As shown in [5], satisfiability is decidable for PDL. Proofs for this result usually rely on model-theoretic properties of PDL, e.g., the small model property and the tree model property.

In order to capture more complex programs, several extensions of PDL have been considered. One can allow new programs operators as, e.g., *converse* [14], or consider an intersection operator on programs to express concurrency properties. Recently, Lutz has shown that PDL with both intersection and converse is decidable [11], extending a difficult result from Danecki [4]. One of the main difficulties when considering such extensions is that they do no longer have the tree model property.

Other extensions use non-regular programs to capture recursive behaviours [8, 9, 6]. Consider the following recursive program [7]:

```
proc V { if p then  {a; call V; b} else return }
```

⋆ Supported by the European Community Research Training Network "Games and Automata for Synthesis and Validation" (Games). Most of this work was done when the second author was a postdoctoral researcher at RWTH Aachen.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 292–306, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The set of executions of this program can be described by the set $\{(p?a)^i(\neg p)?b^i \mid i \geq 0\}$, which is not regular and hence cannot be captured by standard PDL. To overcome this weakness various extensions of PDL by sets of non-regular programs have been considered.

For a class $\mathcal{C}$ of languages over the set of atomic programs as letters, one can distinguish the weak and the strong extension ([6]) of PDL by programs in $\mathcal{C}$. The weak extension allows formulas $\langle\alpha\rangle\varphi$ where α is (a placeholder for) some language from $\mathcal{C}$, whereas in the strong extension the set of atomic programs is augmented by symbols for the languages from $\mathcal{C}$ (which can then be used in the regular expressions). For the weak extension of PDL by the class of context-free languages satisfiability is easily seen to be undecidable. Nevertheless, several strong extensions by single context-free languages are known to be decidable, e.g., the one by $\{a^ib^i \mid i \geq 0\}$, even if they no longer have the finite model property. Surprisingly, the weak extension of PDL with $\{a^iba^i \mid i \geq 0\}$ leads to undecidability. In order to establish the borderline between decidable and undecidable extensions of PDL by non-regular languages, restrictions on pushdown automata (and hence subclasses of context-free languages) have been considered: the largest class of languages, for which the strong extension of PDL is decidable, is the one of deterministic semi simple-minded languages considered in [6]. A pushdown automaton is semi simple-minded if the input letter determines whether to execute a pop, no stack operation, or a push according to a fixed partition of the input alphabet. In case of a push also the stack symbol to be pushed is fixed by the input letter.

A related but stronger subclass of context-free languages that has recently been defined in [3] is the class of visibly pushdown languages. The definition of visibly pushdown automata (VPAs) is the same as for semi simple-minded pushdown automata except that the stack symbol that is pushed may also depend on the current control state and not only on the input letter. It is not difficult to see that this additional freedom indeed allows to define more languages.

We define *recursive PDL* by replacing regular expressions as the formalism to define programs in PDL by VPAs. This extension of PDL with VPAs is more general in two senses than the strong extension of PDL with (semi) simple-minded pushdown automata as considered in [8] and [6]: the model of VPA is more expressive than the model of (semi) simple-minded pushdown automaton, and we do not just augment the set of atomic programs by placeholders for VPA languages (as in the strong extension) but use VPAs as formalism for defining programs. This second property allows not just to use VPAs over atomic programs but also to use tests inside the VPAs.

Our main result is that satisfiability for recursive PDL is decidable in doubly exponential time. To our knowledge, this captures all previous known results concerning decidable extensions of PDL using context-free languages. Futhermore, our proofs are simpler and less technical than the ones in [8, 6] because we can use general resutls and constructions from the theory of VPAs.

The remainder of the paper is organised as follows. In Section 2 we give the basic definitions and results concerning PDL and VPAs, and we define recursive

PDL. In Section 3 we prove the decidability of the satisfiability problem for recursive PDL, and in Section 4 we extend the results to infinite computations.

2 Definitions

In this section, we first define propositional dynamic logic (PDL) using regular programs. Then we introduce visibly pushdown automata and use them to extend PDL with more powerful programs.

Formulas of propositional dynamic logic [5] are interpreted over transition systems whose edges are labelled with atomic programs or actions and whose states are labelled with atomic propositions. Hence, we fix a set $\mathbb{P}$ of *atomic propositions* and a set Π of *atomic programs*. The set of *formulas* and the set of *programs* are defined inductively as follows:

(1) $\top$ is a formula.
(2) Every atomic proposition is a formula.
(3) If φ_1 and φ_2 are formulas, then so are $\neg\varphi_1$ and $\varphi_1 \wedge \varphi_2$.
(4) If φ is a formula, then $\varphi?$ is a *test*. The set of tests is denoted by Test.
(5) If α is a program and φ is a formula, then $\langle\alpha\rangle\varphi$ is a formula. Such a formula will be called a *diamond formula*. The negation of a diamond formula will be called a *box formula*.
(6) A regular expression over $\Pi \cup \text{Test}$ is a program.

In this definition, we refer to standard regular expressions α built from single letters using concatenation, union, and Kleene-star. By $L(\alpha)$ we denote the set of words defined by the regular expression α.

PDL formulas are interpreted over structures $M = (S, R, \nu)$ where S is a set of states, $R : \Pi \to 2^{S\times S}$ is a transition relation, and $\nu : S \to 2^{\mathbb{P}}$ assigns truth values to each atomic proposition in $\mathbb{P}$ for each state in S. In the following, we extend the relation R to all programs and tests, and in parallel define when a formula φ is satisfied in a state s of the structure M, denoted as usual by $M, s \models \varphi$:

- $R(\varphi?) = \{(s, s) \mid M, s \models \varphi\}$ for a test $\varphi?$.
- $R(\alpha)$ for a program α contains the pairs (s, s') for which there are
  - a word $w = w_1 \cdots w_m \in L(\alpha)$ (with $w_i \in \Pi \cup \text{Test}$), and
  - states $s_0, \ldots, s_m \in S$ with $s = s_0$, $s' = s_m$, and $(s_{i-1}, s_i) \in R(w_i)$ for all $1 \leq i \leq m$.
- $M, s \models \varphi_1 \wedge \varphi_2$ if $M, s \models \varphi_1$ and $M, s \models \varphi_2$.
- $M, s \models \neg\varphi$ if $M, s \models \varphi$ does not hold.
- $M, s \models \langle\alpha\rangle\varphi$ if there exists a state s' such that $(s, s') \in R(\alpha)$ and $M, s' \models \varphi$.

A formula φ is *satisfiable* if there is a structure M and a state s such that $M, s \models \varphi$. The *satisfiability problem* is to determine, given a formula φ, whether it is satisfiable.

To show decidability of the satisfiability problem we use tree structures as defined in the following. Let $\Pi = \{a_0, \ldots, a_{n-1}\}$ be a finite set of atomic programs. A *tree structure* for Π is a structure $M = (S, R, \nu)$ such that for some $k \in \mathbb{N}$

- $S \subseteq [k]^*$ is a non-empty, prefix closed set (with $[k] = \{0, \ldots, k-1\}$), and
- $R(a_\ell) = \{(x, xd) \in S \times S \mid x \in [k]^* \text{ and } \ell = d \mod n\}$.

For $x \in [k]^*$ and $d \in [k]$ we call xd the d-successor of x. The second item in the above definition simply states that the relations for the atomic programs are obtained by taking the number of the successor modulo n.

In the following, we introduce a subclass of pushdown automata and consider the logic obtained when replacing regular expressions by this kind of automata for defining programs in PDL.

A pushdown automaton is called visibly pushdown automaton [3], if the type of operation that is performed on the stack, i.e. push, skip, or pop, only depends on the input symbol. For such an automaton one can partition the input alphabet into three sets, consisting of the symbols that induce a push, a skip, or a pop, respectively. In [2] these automata are used to solve verification problems for recursive state machines. In this setting pushes correspond to calls of procedures, skips correspond to internal actions, and pops correspond to returns from procedures. This is where the notation used in the following arises from.

A pushdown alphabet is a tuple $\widetilde{A} = \langle A_c, A_r, A_{\text{int}} \rangle$ consisting of three disjoint finite alphabets that can be interpreted as a finite set of *calls* (A_c), a finite set of *returns* (A_r), and a finite set of *internal actions* (A_{int}). For any such $\widetilde{A}$, let $A = A_c \cup A_r \cup A_{\text{int}}$.

A *visibly pushdown automaton* (VPA) over $\langle A_c, A_r, A_{\text{int}} \rangle$ is a tuple $\mathcal{A} = (Q, \Gamma, Q_{\text{in}}, \delta, F)$ where Q is a finite set of states, $Q_{\text{in}} \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, Γ is a finite stack alphabet that contains a special bottom-of-stack symbol $\perp$ and $\delta \subseteq (Q \times A_c \times Q \times (\Gamma \setminus \{\perp\})) \cup (Q \times A_r \times \Gamma \times Q) \cup (Q \times A_{\text{int}} \times Q)$ is the transition relation.

A *configuration* of $\mathcal{A}$ is a pair $(q, \sigma) \in Q \times (\Gamma \setminus \{\perp\})^* \perp$ of a state q and a *stack content* σ. Note that the symbol $\perp$ may only appear at the bottom of the stack. We denote the set of all configurations of $\mathcal{A}$ by $Cf(\mathcal{A})$.

For a letter $a \in A$, a configuration (q', σ') is an *a-successor* of (q, σ), denoted by $(q, \sigma) \xrightarrow{a} (q', \sigma')$, if one of the following holds:

- For $a \in A_c$, $\sigma' = \gamma\sigma$ and there is a transition $(q, a, q', \gamma) \in \delta$.
- For $a \in A_{\text{int}}$, $\sigma' = \sigma$ and there is a transition $(q, a, q') \in \delta$.
- For $a \in A_r$, either $\sigma = \gamma\sigma'$ and there is a transition $(q, a, \gamma, q') \in \delta$, or $\sigma = \sigma' = \perp$ and there is a transition $(q, a, \perp, q') \in \delta$.

For a finite word $u = a_0 a_1 \cdots a_n$ in A^*, a run of $\mathcal{A}$ on u is a sequence $\rho = (q_0, \sigma_0)(q_1, \sigma_1) \cdots (q_{n+1}, \sigma_{n+1})$ of configurations with $q_0 \in Q_{in}$, $\sigma_0 = \perp$, and $(q_i, \sigma_i) \xrightarrow{a_i} (q_{i+1}, \sigma_{i+1})$ for every $i \in \{1, \ldots, n\}$. In this situation we also write $(q_0, \sigma_0) \xrightarrow{u} (q_{n+1}, \sigma_{n+1})$.

A word $u \in A^*$ is accepted by $\mathcal{A}$ if there is a run of $\mathcal{A}$ on u that ends in a configuration (q, σ) with $q \in F$. The language $L(\mathcal{A})$ of a VPA $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$.

As usual, we call a VPA *complete* if for each configuration (q, σ) and each input symbol a there is at least one a-successor of (q, σ). A VPA is *deterministic* if it has a unique initial state q_{in}, and for each input letter and configuration

there is at most one successor configuration. For deterministic VPAs we write $\delta(q,a) = (q',\gamma)$ instead of $(q,a,q',\gamma) \in \delta$ if $a \in A_c$, $\delta(q,a,\gamma) = q'$ instead of $(q,a,\gamma,q') \in \delta$ if $a \in A_r$, and $\delta(q,a) = q'$ instead of $(q,a,q') \in \delta$ if $a \in A_{\text{int}}$.

One can easily show that visibly pushdown languages are closed under union and intersection using ordinary product constructions. The closure under complement follows from a more complicated construction for determinisation.

Theorem 1 ([3]). *For each* VPA *there is an equivalent deterministic* VPA *of exponential size.*

We need two extensions of VPAs: to infinite words and to infinite trees. For nondeterministic automata, the extension to infinite words is straightforward ([3]). A run on an infinite input word is a sequence of configurations that satisfies the conditions as given in the definition of runs on finite words. The set F of final states is now interpreted as a set of Büchi states, i.e., an infinite run is accepting if it infinitely often visits a configuration with a state from F. We call such automata *nondeterministic Büchi* VPA*s*. If we do not want to explicitly specify the acceptance condition of a VPA on infinite words, then we call it an ω-VPA.

For deterministic automata, the situation is a bit different. In [3] it is shown that the standard acceptance conditions do not suffice to obtain a deterministic model that is as expressive as nondeterministic Büchi VPAs. We can avoid this problem if we evaluate the acceptance condition on a certain subsequence of the run [10]. This leads to the model of stair parity VPAs.

A *stair parity* VPA over $\widetilde{A}$ is of the form $\mathcal{A} = (Q, \Gamma, Q_{\text{in}}, \delta, \Omega)$ where Q, Γ, Q_{in} and δ are as in VPAs and $\Omega : Q \to \mathbb{N}$ is a priority function. To define acceptance for stair parity VPAs we first have to filter out the relevant positions in a run. Let $\rho = (q_0, \sigma_0)(q_1, \sigma_1)\cdots$ be an infinite run of $\mathcal{A}$. For $i \in \mathbb{N}$ we call (q_i, σ_i) a *step* of ρ if in all successive positions the height of stack does not go below the height of σ_i, i.e., $|\sigma_j| \geq |\sigma_i|$ for all $j \geq i$.

Note that, since the height of the stack at each position solely depends on the input, the set of positions of the steps is the same for different runs on the same input word.

The run $\rho = (q_0, \sigma_0)(q_1, \sigma_1)\cdots$ is accepting if the maximal priority that occurs infinitely often on a step is even, i.e., if

$$\max\{\Omega(q) \mid q = q_i \text{ for infinitely many steps } (q_i, \sigma_i) \text{ of } \rho\}$$

is even. The definition of deterministic stair parity VPA is directly adapted from the definition of deterministic VPA.

Theorem 2 ([10]). *For each nondeterministic Büchi* VPA *there exists a deterministic stair parity* VPA *recognising the same language.*

We also need two very simple acceptance conditions for VPAs on infinite words: reachability and safety. Both conditions are specified by a set F of states. A run of a reachability VPA is accepting if some state from F occurs in this run. Dually, a run of a safety VPA is accepting if no state from F occurs in the run.

Obviously, a deterministic safety VPA accepts the complement of the language accepted by the same automaton viewed as a reachability VPA.

If a reachability VPA $\mathcal{A}$ is complete, then the accepted language is of the form $L \cdot A^\omega$ for the language L accepted by $\mathcal{A}$ viewed as a VPA on finite words. Hence, we obtain the following corollary to Theorem 1.

Corollary 1. *For each complete nondeterministic reachability* VPA *there exists an equivalent deterministic reachability* VPA *of exponential size.*

To define visibly pushdown tree automata we consider infinite k-ary Σ-labelled trees, i.e., mappings $t : [k]^* \to \Sigma$. By $\mathcal{T}_{k,\Sigma}$ we denote the set of all infinite k-ary Σ-labelled trees.

The setting on trees that we need is slightly different from the word case: the stack operation performed in a transition of a tree automaton is not determined by the node label but by the direction in the tree. Hence, we assume that $A = [k]$.

A visibly pushdown tree automaton (VPTA) over $\widetilde{A}$ (with $A = [k]$) is of the form $\mathcal{A} = (Q, \Sigma, \Gamma, Q_{\text{in}}, \delta, Acc)$ where Q, Γ, Q_{in} are as for VPAs on words, Σ is a node label alphabet, Acc is the acceptance component, and δ is the set of transitions. A transition is of the form (q, b, γ, τ) with $q \in Q$, $b \in \Sigma$, $\gamma \in \Gamma$, and $\tau : [k] \to Q \cup (Q \times \Gamma)$ such that $\tau(d) \in Q$ if $d \in A_{\text{int}} \cup A_r$ and $\tau(d) \in Q \times \Gamma$ if $d \in A_c$. A configuration of $\mathcal{A}$ is defined as before.

For a tree $t : [k]^* \to \Sigma$, a run of $\mathcal{A}$ on t is a mapping $\rho : [k]^* \to Cf(\mathcal{A})$ such that $\rho(\varepsilon) \in Q_{\text{in}} \times \{\bot\}$ is an initial configuration, and for each $x \in [k]^*$ with $\rho(x) = (q, \gamma\sigma)$ there is a transition $(q, t(x), \gamma, \tau) \in \delta$ such that for all $d \in A$:

$$\rho(xd) = \begin{cases} (q', \gamma\sigma) & \text{if } d \in A_{\text{int}} \text{ and } \tau(d) = q', \\ (q', \sigma) & \text{if } d \in A_r,\ \tau(d) = q',\ \text{and } \gamma \in \Gamma \setminus \{\bot\}, \\ (q', \bot) & \text{if } d \in A_r,\ \tau(d) = q', \sigma = \varepsilon,\ \text{and } \gamma = \bot, \\ (q', \gamma'\gamma\sigma) & \text{if } d \in A_c \text{ and } \tau(d) = (q', \gamma'). \end{cases}$$

Intuitively, if the automaton is at a certain node of the input tree, it reads the label of the node and then sends copies of itself to all the successors of the node. Depending on the type of the successor (call, return, or internal action) the automaton performs a push, a pop, or leaves the stack unchanged.

As for VPAs, we can consider different types of acceptance for VPTAs, e.g., Büchi, parity, or stair parity conditions with the corresponding component substituted for Acc. Then $\mathcal{A}$ accepts an input tree t if there is a run of $\mathcal{A}$ on t such that each path through this run (which is an infinite sequence of configurations) satisfies the acceptance condition. The set of all trees accepted by $\mathcal{A}$ is denoted by $\mathcal{T}(\mathcal{A})$.

Similar to the case of finite automata on infinite trees (cf. [15]), the emptiness test for a VPTA is polynomial time equivalent to the problem of determining the winner in a visibly pushdown game ([10]) with a winning condition corresponding to the acceptance condition of the VPTA. Since solving such games is complete for exponential time (for all the winning conditions considered here), we obtain the following theorem (and also corresponding lower bounds).

Theorem 3. *For a given* VPTA $\mathcal{A}$ *one can decide in exponential time whether* $\mathcal{T}(\mathcal{A})$ *is empty.*

For later use, we need to relate VPAs on words and on trees. For this purpose, we code paths through k-ary Σ-labelled trees by words that can be processed by a VPA.

An infinite path can be uniquely identified with an infinite sequence $d_0 d_1 d_2 \cdots$ with $d_i \in [k]$. Given such a path π and a tree $t : [k]^* \to \Sigma$, we define the infinite word $w_\pi^t \in (\Sigma \times [k])^\omega$ as $w_\pi^t = (t(\varepsilon), d_0)(t(d_0), d_1)(t(d_0 d_1), d_2) \cdots$.

The partition of the alphabet $\Sigma \times [k]$ into calls, returns, and internal actions is inherited from the partition of $[k]$.

For a language $L \subseteq (\Sigma \times [k])^\omega$ we define the corresponding language of trees that contains exactly those trees for which all codings of paths are in L:

$$\text{Trees}(L) = \{t \in \mathcal{T}_{k,\Sigma} \mid w_\pi^t \in L \text{ for all paths } \pi\}.$$

If L is accepted by some deterministic ω-VPA $\mathcal{A}$, then one can easily define a VPTA accepting Trees(L) by simulating $\mathcal{A}$ on each path.

Remark 1. For each deterministic ω-VPA $\mathcal{A}$ over $\Sigma \times [k]$ there exists a VPTA with an acceptance condition of the same type accepting Trees($L(\mathcal{A})$).

The formalism of recursive PDL is obtained by replacing regular expressions (as the formalism to define programs) by VPAs. For this purpose we assume that the set of atomic programs is given as a pushdown alphabet $\widetilde{\Pi} = \langle \Pi_c, \Pi_{int}, \Pi_r \rangle$ of calls Π_c, internal actions Π_{int}, and returns Π_r as required for VPAs. The set of formulas of recursive PDL is defined in the same way as for PDL. To define the set of programs we replace (6) from the syntax definition of PDL by

(6′) A VPA $\mathcal{A}$ over $\langle \Pi_c, \Pi_{int} \cup \text{Test}, \Pi_r \rangle$ is a program.

So we replace regular expressions or finite automata by VPAs, where tests are treated as internal actions. Note that an atomic program a may be seen as a singleton $\{a\}$ and thus as a visibly pushdown language. Therefore, we will always assume that all diamond formulas are of the form $\langle \mathcal{A} \rangle \varphi$ for some VPA $\mathcal{A}$.

The definition of the semantics does not change. The only difference is that in the extension of the relation R to programs we now refer to the language defined by VPAs instead of regular expressions.

Example 1. Consider the set of atomic programs given by $\widetilde{\Pi} = \langle \{c_0, c_1\}, \emptyset, \{r_0, r_1\} \rangle$ and the set $\mathbb{P} = \{p_0, p_1\}$ of atomic propositions. Let

- $\psi = \langle \mathcal{B} \rangle p_0$ where $\mathcal{B}$ accepts the language $\{c_1^k r_1^k \mid k > 0\}$, and
- $\varphi = \langle \mathcal{A} \rangle p_1$ where $\mathcal{A}$ accepts the language $\{((\psi?)c_0)^k r_0^k \mid k > 0\}$.

For the structure M, as depicted in Figure 1 with $p_1 \in \nu(s_1)$ and $p_0 \in \nu(s_1')$, we have $(s, s_1') \in R(\mathcal{B})$ and $(s', s_1') \in R(\mathcal{B})$. Since $p_0 \in \nu(s_1')$, we obtain $M, s \models \psi$ and $M, s' \models \psi$. Thus, $(s, s), (s', s') \in R(\psi?)$ and therefore $(s, s_1) \in R(\mathcal{A})$. Since $p_1 \in \nu(s_1)$ we finally obtain that $M, s \models \varphi$.

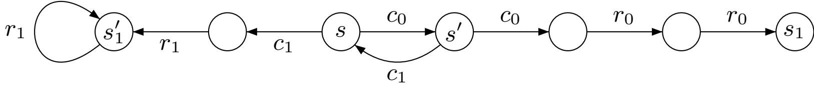


Fig. 1. A model (with s as initial state) for the formula from Example 1

Note that this extension of PDL with VPAs is more general in two senses than the extension of PDL with (semi) simple-minded pushdown automata considered in [8] and [6]. First, VPAs are more expressive than semi simple-minded pushdown automata as, for example, witnessed by the language containing the words of the form $c^n c^m r_1^m c^\ell r_2^\ell r_1^n$ (for c being a call and r_1, r_2 being returns). The proof of this is straightforward because a semi simple-minded pushdown automaton with only one call symbol can only use a single stack symbol.

Second, VPAs can be nested in recursive PDL by using tests as in Example 1. If we view VPAs as descriptions of recursive procedures, then this nesting allows, for example, not only to test properties on entering and exiting a procedure but also to relate these properties to tests that are "launched" inside the execution of the procedure. A simple example for such a formula is $\langle p_1?; c^n; p_2?; r^n; p_3\rangle\varphi$, where the numbers of calls before, and returns after the test $p_2?$ have to be the same. This is not possible in the extensions of PDL considered in [8, 6], where the non-regular languages that are used as programs are languages over atomic programs only.

3 Satisfiability for Recursive PDL

In this section, we show that the satisfiability problem for recursive PDL is decidable. The idea for the satisfiability test is the same as in [16] and [8]. One first shows that each recursive PDL formula φ has a tree model. In these tree models one labels each node s with all subformulas of φ that are true in s. Such trees are called Hintikka trees. Then one constructs a tree automaton that accepts the Hintikka trees of φ and checks this automaton for emptiness. When starting with a PDL formula one obtains a Büchi tree automaton. Since we use VPAs for the definition of programs we will end up with a visibly pushdown tree automaton.

The following definitions and propositions concerning Hintikka trees are simple adaptions from [8], we just recall them here for completeness.

From now on, we identify a formula φ with the formula $\neg\neg\varphi$. For each formula φ in recursive PDL, we define its closure $\mathrm{cl}(\varphi)$ as the minimal set satisfying the following:

- $\varphi \in \mathrm{cl}(\varphi)$.
- If $\varphi_1 \wedge \varphi_2 \in \mathrm{cl}(\varphi)$, then $\varphi_1, \varphi_2 \in \mathrm{cl}(\varphi)$.
- If $\psi \in \mathrm{cl}(\varphi)$, then $\neg\psi \in \mathrm{cl}(\varphi)$.
- If $\langle\mathcal{A}\rangle\psi \in \mathrm{cl}(\varphi)$, then $\psi \in \mathrm{cl}(\varphi)$. Additionally, if $\psi'?$ is an internal action in $\mathcal{A}$, then $\psi' \in \mathrm{cl}(\varphi)$.

Note that the size of $\mathrm{cl}(\varphi)$ is linear in the size of φ. By $\mathrm{cl}_\diamond(\varphi)$ we denote the set of all diamond formulas from $\mathrm{cl}(\varphi)$.

We now fix a formula φ of recursive PDL containing n atomic programs $a_0, \dots, a_{n-1}$. Furthermore, we assume w.l.o.g. that all atomic propositions from $\mathbb{P}$ are used in φ.

A tree structure $M = (S, R, \nu)$ is a *tree model* for φ if $M, \varepsilon \models \varphi$. As for PDL formulas, one can show that if a recursive PDL formula has a model then it has a tree model.

Proposition 1. *A formula of recursive PDL is satisfiable if and only if it has a tree model.*

We now define the notion of *Hintikka tree.* For this purpose we define the alphabet $\Sigma_\varphi = 2^{\mathrm{cl}(\varphi)} \cup \{\bot\}$, where $\bot$ is some symbol used to label nodes that do not have to be considered. Note that this use of $\bot$ is not at all related to the bottom-of-stack symbol used for VPAs.

Definition 1. *A* Hintikka tree *for a formula φ of recursive PDL with atomic programs $a_0, \dots, a_{n-1}$ is a k-ary tree $t : [k]^* \to \Sigma_\varphi$ with $k \geq n$ such that $\varphi \in t(\varepsilon)$, and for all elements $x \in [k]^*$:*

1. *If $t(x) \neq \bot$, then $\psi \in t(x)$ if and only if $\neg\psi \notin t(x)$ for all $\psi \in \mathrm{cl}(\varphi)$.*
2. *If $\psi_1 \wedge \psi_2 \in \mathrm{cl}(\varphi)$, then $\psi_1 \wedge \psi_2 \in t(x)$ if and only if $\psi_1, \psi_2 \in t(x)$.*
3. (Diamond property) *$\langle\mathcal{A}\rangle\psi \in t(x)$ if and only if there exists an $\mathcal{A}$-path (to be defined below) from x to y in t for some $y \in [k]^*$ such that $\psi \in t(y)$.*
4. (Box property) *$\neg\langle\mathcal{A}\rangle\psi \in t(x)$ if and only if $\psi \notin t(y)$ for all $y \in [k]^*$ for which there is an $\mathcal{A}$-path from x to y.*

An $\mathcal{A}$-path *from a node x to a node y is a sequence $x_0, \dots, x_m$ of nodes with $x_0 = x$ and $x_m = y$ such that there is a word $w = w_1 \cdots w_m \in L(\mathcal{A})$ and the following holds for all $i = 1, \dots, m$:*

- *If $w_i = \psi'?$ for some formula ψ', then $x_i = x_{i-1}$ and $\psi' \in t(x_{i-1})$.*
- *If $w_i = a_\ell$ for some atomic program a_ℓ, then $x_i = x_{i-1}d$ for some d with $\ell = d \mod n$.*

The $\mathcal{A}$-path required in 3 of the previous definition is also called a *witnessing path* for $\langle\mathcal{A}\rangle\psi$.

It is not difficult to see that Hintikka trees for φ are obtained from tree models of φ by annotating each node with the set of formulas that are satisfied in this node.

Proposition 2. *Let φ be a formula of recursive PDL. There is a Hintikka tree for φ if and only if φ has a tree model.*

Our goal is to build a tree automaton that accepts Hintikka trees for φ. Such an automaton has to verify for each node x with a diamond formula $\langle\mathcal{A}\rangle\psi$ in $t(x)$ that there is an $\mathcal{A}$-path starting from x to some node y. Such paths may overlap and the tree automaton would have to keep track of which VPAs to simulate in order to check the diamond property for several nodes. To simplify this task we show that it is always possible to find a Hintikka tree where the

paths witnessing the diamond properties are (edge) disjoint. Such Hintikka trees are called *unique diamond path Hintikka trees* in [8]. In the definition from [8] it is possible that for a diamond formula $\langle\mathcal{A}\rangle\psi$ that is in $t(x)$ the witnessing path contains a node y such that $\langle\mathcal{A}\rangle\psi$ is also in $t(y)$. Then the witnessing path for this second occurrence of the diamond formula might overlap the witnessing path for the first occurrence. In our definition we also avoid this problem.

Definition 2. *A* unique diamond path Hintikka tree *for φ is a Hintikka tree t for φ that satisfies the following additional condition: there exists a mapping $\rho : [k]^* \to (\mathrm{cl}_\diamond(\varphi) \times [k]^*) \cup \{\bot\}$, such that for all $x \in [k]^*$: If $\langle\mathcal{A}\rangle\psi \in t(x)$ then, for some witnessing $\mathcal{A}$-path $x_0, \ldots, x_m$ (starting in x), we have $\rho(x_i) = (\langle\mathcal{A}\rangle\psi, x)$ for all $1 \leq i \leq m$.*

Any Hintikka tree can be transformed into a unique diamond path Hintikka tree by increasing the number of descendants of each node such that there is a separate branch for each formula when needed. The branching degree resulting from this increase of descendants can be bounded as stated in the following proposition, where r denotes the number of diamond formulas in $\mathrm{cl}(\varphi)$ and n the number of atomic programs.

Proposition 3. *Let φ be a formula of recursive PDL. There is a Hintikka tree for φ if and only if there is a k-ary unique diamond path Hintikka tree for φ with $k = 2^{|\mathrm{cl}(\varphi)|} \cdot n \cdot 2r$.*

We now show how to build a Büchi VPTA accepting exactly the k-ary unique diamond path Hintikka trees for φ. Together with Theorem 3 one obtains decidability of the satisfiability problem for recursive PDL formulas.

So from now on we are interested in trees from $\mathcal{T}_{k,\Sigma_\varphi}$. Further, note that each $d \in [k]$ is associated in a natural way to an atomic program in the definition of $\mathcal{A}$-path, namely to a_ℓ if $\ell = d \mod n$. This directly induces a partition of $[k]$ into calls, returns, and internal actions.

The construction of the VPTA follows the same lines as in [8]. We first build three visibly pushdown tree automata. The first automaton is called the *local automaton* and accepts all trees satisfying the first two items of Definition 1. The second automaton called *box automaton* accepts all trees satisfying the box property (see Definition 1). The third automaton called *diamond automaton* accepts all trees satisfying the diamond property (see Definition 1) and the condition of Definition 2.

The intersection of the languages accepted by these three automata defines exactly the set of k-ary unique diamond path Hintikka trees for φ. As visibly pushdown tree languages are closed under intersection, a nondeterministic visibly pushdown tree automaton recognising the desired language can be constructed.

Local automaton. The local automaton is easily constructed as a two-state finite tree automaton equipped with a safety condition. The automaton checks for all nodes x in the tree whether $t(x)$ satisfies the first two conditions of Definition 1. If in some node one of these two conditions is violated, the automaton goes to its rejecting state, otherwise it stays in the initial state.

Lemma 1. *There is a finite tree automaton with a safety acceptance condition and two states that accepts the trees that satisfy the first two properties of Definition 1.*

Box automaton. We now construct a VPTA accepting those trees from $\mathcal{T}_{k,\Sigma_\varphi}$ that satisfy the box property from Definition 1. First note that the box property is a condition on the paths through the tree. This means we can define a language $L_{\text{box}} \subseteq (\Sigma_\varphi \times [k])^\omega$ such that $T_{\text{box}} = \text{Trees}(L_{\text{box}})$, where T_{box} denotes the set of all trees satisfying the box property. We now define L_{box} and then show that it can be accepted by a deterministic safety VPA.

For each word $w \in (\Sigma_\varphi \times [k])^\omega$ there exists a tree $t \in \mathcal{T}_{k,\Sigma_\varphi}$ and a path π such that $w = w^t_\pi$. Then w is in L_{box} if this t satisfies the box property on π: for all $x \in \pi$, $\neg\langle\mathcal{A}\rangle\psi \in t(x)$ if and only if $\psi \notin t(y)$ for all $y \in \pi$ for which there is an $\mathcal{A}$-path from x to y.

It is not difficult to see that $t \in \mathcal{T}_{k,\Sigma_\varphi}$ indeed satisfies the box property if and only if all its paths are in L_{box}. Hence, by Remark 1, to construct a VPTA for T_{box} it is sufficient to construct a deterministic VPA for L_{box}.

Lemma 2. *There is a deterministic safety* VPA *of size exponential in the size of φ that accepts L_{box}.*

Proof. Let $\psi_1, \ldots, \psi_m$ be an enumeration of all box formulas $\psi_i = \neg\langle\mathcal{A}_i\rangle\varphi_i \in \text{cl}(\varphi)$. We show how to construct a visibly pushdown automaton for the complement $\overline{L}_{\text{box}}$ of L_{box}, and we conclude using closure of visibly pushdown languages under complementation.

First note that $\overline{L}_{\text{box}} = \bigcup_{i=1}^m \overline{L}_i$, where $\overline{L}_i$ is the set of all words describing a path that violates the box condition for ψ_i. For every i, $\overline{L}_i$ is accepted by a VPA $\mathcal{B}_i$ equipped with a reachability condition as follows.

For an input word $w = (C_0, d_0)(C_1, d_1)\cdots$ with $C_j \in \Sigma_\varphi$ and $d_j \in [k]$ the VPA $\mathcal{B}_i$ guesses a segment $(C_j, d_j)\cdots(C_{j'}, d_{j'})$ with $\psi_i \in C_j$ and $\varphi_i \in C_{j'}$, and verifies that it corresponds to an $\mathcal{A}_i$-path. This is realised as follows:

- Before guessing the initial position j of the segment, $\mathcal{B}_i$ stores a special symbol $\sharp$ on the stack. On guessing j it enters a state indicating that the simulation of $\mathcal{A}_i$ starts.
- In the simulation phase, on reading a letter (C, d), $\mathcal{B}_i$ can simulate a sequence of transitions of $\mathcal{A}_i$ consisting of tests and ending with the atomic program a_ℓ corresponding to d, i.e., with $\ell = d \mod n$. So, a change of configuration in $\mathcal{A}_i$ on reading a word of the form $\chi_1? \cdots \chi_r?a_\ell$ is performed in $\mathcal{B}_i$ in a single transition on (C, d) if $\chi_1, \ldots, \chi_r$ are in $C \neq \bot$. This is possible since tests are handled as internal actions in $\mathcal{A}_i$ and thus only induce a change of the control state.

  In this simulation, whenever $\mathcal{B}_i$ sees $\sharp$ as top stack symbol, it treats it as the bottom-of-stack symbol $\bot$ is handled in $\mathcal{A}_i$.
- Finally, if $\mathcal{B}_i$ reads (C, d) with $\varphi_i \in C$, and there is a (possibly empty) sequence $\chi_1? \cdots \chi_r?$ of tests leading to an accepting state in $\mathcal{A}_i$ where χ_1, $\ldots, \chi_r$ are in C, then $\mathcal{B}_i$ can move to its accepting state on reading (C, d). Once $\mathcal{B}_i$ has reached its accepting state it remains there forever.

Note that the size of $\mathcal{B}_i$ is linear in the size of $\mathcal{A}_i$. Furthermore, $\mathcal{B}_i$ can be constructed such that it is complete because every run that reaches an accepting state never stops.

Taking the union of these VPAs one obtains a reachability VPA $\mathcal{B}$ for $\overline{L}_{\text{box}}$. Determinising and then complementing $\mathcal{B}$ (see Corollary 1) yields a safety VPA for L_{box} that is of size exponential in $\mathcal{B}$ and thus also exponential in the size of φ. □

Applying Remark 1 we directly get the following result.

Lemma 3. *There is a safety* VPTA *of size exponential in the size of φ that accepts T_{box}.*

Diamond automaton. We give an informal description of the diamond automaton. This automaton is designed to accept trees that satisfy both the diamond condition and the one of Definition 2.

The control state of the diamond automaton stores the following informations:

- A diamond formula $\langle\mathcal{A}\rangle\psi$ currently checked or $\perp$ if nothing is checked.
- If some diamond formula $\langle\mathcal{A}\rangle\psi$ is being checked, a control state of $\mathcal{A}$ is stored (and stack information from $\mathcal{A}$ will be encoded in the stack of the diamond automaton).

At the beginning no formula is checked. The diamond automaton reads the labelling $t(x)$ of the current node x. If it contains some diamond formula, it will go for each of these formulas in a different branch of the tree where it checks this formula. If the automaton was already checking for a diamond formula, it keeps looking for its validation by choosing yet another branch. As the tree should satisfy the unique diamond path property, a validation of the diamond formulas can be found in this way.

When checking for a diamond formula $\langle\mathcal{A}\rangle\psi$, the automaton performs a simulation of $\mathcal{A}$ on the path it guesses. A sequence of tests read by $\mathcal{A}$ followed by some atomic program is simulated in a single transition of the VPTA. For this it stores in its control state the current state q of $\mathcal{A}$ in the simulation and uses its stack to mimic the one of $\mathcal{A}$. Assume that in $\mathcal{A}$ a sequence of the following form is possible: $(q,\gamma) \xrightarrow{\chi_1?} (q_1,\gamma) \xrightarrow{\chi_2?} \cdots \xrightarrow{\chi_m?} (q_m,\gamma) \xrightarrow{a_\ell} (q',\sigma)$, where γ denotes the top stack symbol and σ is the new top of the stack, depending on the type of a_ℓ, i.e., $\sigma = \varepsilon$ for a return, $\sigma = \gamma$ for an internal action, and $\sigma = \gamma'\gamma$ for a call and some γ' from the stack alphabet of $\mathcal{A}$. Then the VPTA on reading a node label $t(x)$ that contains $\chi_1,\ldots,\chi_m$ can update the state q of $\mathcal{A}$ to q' when proceeding to a d-successor with $\ell = d \mod k$.

To keep track of the level of the stack where the simulation of $\mathcal{A}$ started, the first symbol pushed onto the stack after starting the simulation of $\mathcal{A}$ is marked by $\sharp$. If this symbol is popped later, then it is recorded in the state of the VPTA that the simulation is at the bottom of the stack, i.e., $\mathcal{A}$-transitions are simulated as if $\perp$ would be the top stack symbol. If a symbol is pushed, it is again marked by $\sharp$.

The simulation ends if the current node label $t(x)$ contains ψ and from the current state q of the $\mathcal{A}$-simulation a final state of $\mathcal{A}$ is reachable by a (possibly empty) sequence of tests such that the corresponding formulas are included

in $t(x)$. In this case the VPTA signals this successful simulation in the next transition by setting a special flag in all successor states. This flag also defines the acceptance condition. If the flag is set infinitely often on each path, then the input is accepted. For this to work we also set the flag if no simulation is performed. This acceptance condition is of Büchi type and hence we have the following result.

Lemma 4. *There is a Büchi* VPTA *of size $\mathcal{O}(|\varphi|)$ that accepts those trees from $\mathcal{T}_{k,\Sigma_\varphi}$ that satisfy the diamond property and the condition of Definition 2.*

Now, consider the automaton obtained by taking the product of the local automaton, the box automaton, and the diamond automaton. The combination of two safety conditions and one Büchi condition can easily be transformed into a single Büchi condition.

Lemma 5. *There is a Büchi* VPTA *of size exponential in the size of φ that accepts the k-ary unique diamond path Hintikka trees for φ.*

Using Theorem 3 we deduce the decidability of the satisfiability problem for recursive PDL formulas.

Theorem 4. *Given a recursive PDL formula, one can decide in doubly exponential time whether it is satisfiable.*

We leave open the question whether this complexity is optimal. A singly exponential lower bound directly follows from the one for standard PDL [5].

4 Extension to Infinite Computations

In [14] an extension of PDL with a construct $\Delta\alpha$ for building formulas from programs α is considered. The meaning of such a formula is that the program α can be repeated infinitely often. The resulting logic is called Δ-PDL. In this section we extend recursive PDL by a similar construct $\Delta\mathcal{A}$ for Büchi VPAs $\mathcal{A}$ over atomic programs and tests. The meaning of such a formula is that there exists a path that is accepted by $\mathcal{A}$.

For the formal definition we introduce the notion of ω-program and add to the syntax rules of recursive PDL the following clauses:

- A Büchi VPA $\mathcal{A}$ over $\langle \Pi_c, \Pi_{int} \cup \text{Test}, \Pi_r \rangle$ is an ω-program.
- If $\mathcal{A}$ is an ω-program, then $\Delta\mathcal{A}$ is a formula.

This extension is called recursive Δ-PDL. For the semantics we only give the definitions for the new constructs. Each ω-program defines a unary relation R_ω and the corresponding Δ-formulas hold at those states of the structure that are in R_ω:

- $s \in R_\omega(\mathcal{A})$ if and only if there is an infinite word $w = w_0w_1w_2 \cdots \in L(\mathcal{A})$ (with $w_i \in \Pi \cup \text{Test}$) and a sequence $s_0, s_1, s_2, \ldots$ of states of the structure such that $s = s_0$ and $(s_i, s_{i+1}) \in R(w_i)$ for all $i \geq 0$.
- $M, s \models \Delta\mathcal{A}$ if and only if $s \in R_\omega(\mathcal{A})$.

The definition of Hintikka tree extends in a straightforward way by adding the natural properties for formulas $\Delta\mathcal{A}$ and $\neg\Delta\mathcal{A}$. In the following, we call these properties Δ-property and $\neg\Delta$-property. The notion of unique diamond path Hintikka tree has to be extended by also requiring *unique Δ-paths*. One easily shows that (adapted versions of) Propositions 1, 2, and 3 still hold.

Then one can construct a VPTA that accepts all trees that have the Δ-property and unique Δ-paths. This construction is similar to the one of the diamond automaton and results in a Büchi VPTA of size linear in the size of the given formula φ.

For the $\neg\Delta$-property one can proceed in a similar way as for the box property. One defines the word language $L_{\neg\Delta}$ corresponding to L_{box} and shows that this language can be accepted by a deterministic VPA. The main difference here is that instead of obtaining a reachability VPA for the complement of $L_{\neg\Delta}$ we obtain a nondeterministic Büchi VPA. Hence, to get a deterministic VPA for $L_{\neg\Delta}$ we have to use a stair parity condition (Theorem 2). All this results in the following lemma.

Lemma 6. *For every recursive Δ-PDL formula φ there is a stair parity* VPTA *of size exponential in the size of φ accepting the unique diamond path and unique Δ-path Hintikka trees of φ.*

Finally, one has to check emptiness for a stair parity VPTA, which can be done in exponential time (Theorem 3).

Theorem 5. *Given a recursive Δ-PDL formula, one can decide in doubly exponential time whether it is satisfiable.*

Again, we leave open the question whether this complexity is optimal.

5 Conclusion

Using visibly pushdown automata we have defined recursive PDL as an extension of regular PDL that allows to capture the behaviour of recursive programs. The result on the satisfiability of this logic subsumes all known decidable extensions of PDL with context-free programs. Comparisons of recursive PDL with μ-calculus using relational fixed points and with visibly pushdown μ-calculus would be interesting. The first one [13] allows to capture the example from the introduction using the formula $\mu R.((p?; a; R; b) \cup (\neg p)?)$ (for a binary relation symbol R), while the second one [1] embeds in the modal μ-calculus the formalism of visibly pushdown automata. Another possible direction for future research is to combine visibly pushdown automata with the game logic of Parikh [12].

References

1. R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *Proceedings of POPL'06*. To appear.
2. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004.

3. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of STOC'04*, pages 202–211. ACM, 2004.
4. R. Danecki. Nondeterministic propositional dynamic logic with intersection is decidable. In *Proceedings of the 5th Symposium on Computation Theory*, volume 208 of *LNCS*, pages 34–53. Springer, 1984.
5. M.J. Fischer and R.E. Ladner. Propositional dyncamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.
6. D. Harel and M. Kaminsky. Strengthened results on nonregular PDL. Technical Report MCS99-13, Weizmann Institute of Science, 1999.
7. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.
8. D. Harel and D. Raz. Deciding properties of nonregular programs. *SIAM Journal on Computing*, 22(4):857–874, 1993.
9. D. Harel and E. Singerman. More on nonregular PDL: Expressive power, finite models, fibonacci programs. In *ISTCS: 3rd Israeli Symposium on the Theory of Computing and Systems*, 1995.
10. C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *Proceedings of FST&TCS'04*, volume 3328 of *LNCS*, pages 408–420. Springer, 2004.
11. C. Lutz. PDL with intersection and converse is decidable. In *Proceedings of CSL'05*, volume 3634 of *LNCS*, pages 413–427. Springer, 2005.
12. R. Parikh. The logic of games an its applications. *Annals of discrete mathematics*, 24:111–140, 1985.
13. D. Park. Finiteness is μ-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.
14. R. Streett. Propositional dynamic logic of looping and converse is elementary decidable. *Information and Control*, 54:121–141, 1982.
15. W. Thomas. Languages, automata, and logic. In *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer, 1997.
16. M. Vardi and P. Wolper. Automata-theoretic techniques for modal logic of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

A Semantic Approach to Interpolation*

Andrei Popescu**, Traian Florin Şerbănuţă***, and Grigore Roşu

Department of Computer Science,
University of Illinois at Urbana-Champaign
{popescu2, tserban2, grosu}@cs.uiuc.edu

Abstract. Interpolation results are investigated for various types of formulae. By shifting the focus from syntactic to *semantic interpolation*, we generate, prove and classify a series of interpolation results for first-order logic. A few of these results non-trivially generalize known interpolation results. All the others are new.

1 Introduction

Craig interpolation is a landmark result in first-order logic [7]. In its original formulation, it says that given sentences Γ_1 and Γ_2 such that $\Gamma_1 \vdash \Gamma_2$, there is some sentence Γ whose non-logical symbols occur in *both* Γ_1 and Γ_2, called an *interpolant*, such that $\Gamma_1 \vdash \Gamma$ and $\Gamma \vdash \Gamma_2$. This well-known result can also be rephrased as follows: given first-order signatures Σ_1 and Σ_2, a Σ_1-sentence Γ_1 and a Σ_2-sentence Γ_2 such that $\Gamma_1 \models_{\Sigma_1 \cup \Sigma_2} \Gamma_2$, there is some $(\Sigma_1 \cap \Sigma_2)$-sentence Γ such that $\Gamma_1 \models_{\Sigma_1} \Gamma$ and $\Gamma \models_{\Sigma_2} \Gamma_2$.

One naturally looks for this property in logical systems other than first-order logic. The conclusion of studying various extensions of first-order logic was that "interpolation is indeed [a] rare [property in logical systems]" ([2], page 68). We are going to show in this paper that the situation is totally different when one looks in the opposite direction, at restrictions of first-order logic. There are simple logics, such as equational logic, where the interpolation result does *not* hold for sentences, but it holds for *sets of sentences* [27]. For this reason, as well as for reasons coming from theoretical software engineering, in particular from specification theory and modularization [3, 13, 14, 9], it is quite common today to state interpolation more generally, in terms of sets of sentences Γ_1, Γ_2, and Γ. This is also the approach that we follow in this paper.

We call our approach to interpolation "semantic" because we shift the problem of finding syntactic interpolants Γ to a problem of finding appropriate *classes*

* Supported by NSF grants CCF-0234524, CCF-044851, CNS-0509321.

** Also: Institute of Mathematics "Simion Stoilow" of the Romanian Academy, Bucharest; and Fundamentals of Computer Science, Faculty of Mathematics, University of Bucharest.

*** Also: Fundamentals of Computer Science, Faculty of Mathematics, University of Bucharest.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 307–321, 2006.
© Springer-Verlag Berlin Heidelberg 2006

of models, which we call *semantic interpolants*. We present a precise characterization for *all* the semantic interpolants of a given instance $\Gamma_1 \models_{\Sigma_1 \cup \Sigma_2} \Gamma_2$, as well as a general theorem ensuring the existence of semantic interpolants closed under generic closure operators. Not all semantic interpolants correspond to sets of sentences. However, when semantic interpolants are closed under certain operators, they become *axiomatizable*, thus corresponding to some sets of sentences. Following the nice idea of using Birkhoff-like axiomatizability to prove the Craig interpolation for equational logics in [27], a similar semantic approach was investigated in [25], but it was only applied there to obtain Craig interpolation results for categorical generalizations of equational logics. A similar idea is exploited in [9], where interpolation results are presented in an institutional [17] setting. While the institution-independent interpolation results in [9] can potentially be applied to various particular logics, their instances still refer to just one type of sentence: the one the particular logic comes with.

The conceptual novelty of our semantic approach to interpolation in this paper is to keep the restrictions on Γ_1, Γ_2, and Γ, or more precisely the ones on their corresponding classes of models, *independent*. This way, surprising and interesting results can be obtained with respect to the three types of sentences involved. By considering several combinations of closure operators allowed by our parametric semantic interpolation theorem, we provide many interpolation results;[1] some of them generalize known results, but most of them are new. For example, we show that if the sentences in Γ_1 are first-order while the ones in Γ_2 are universally quantified Horn clauses (UHC's), then those in the interpolant Γ can be chosen to be UHC's too. Surprisingly, sometimes the interpolant is strictly simpler than Γ_1 and Γ_2. For example, we show that the following choices of the type of sentences in the interpolant Γ are possible (see also the table on page 316, lines 6, 13 and 20):

- Γ_1-universal and Γ_2-positive (i.e., contains only formulae without negations) imply that Γ consists only of universally quantified disjunction of atoms;
- Γ_1-UHC's and Γ_2-positive imply that Γ has only universally quantified atoms;
- Γ_1- finitary formulae and Γ_2- infinitary universally quantified disjunctions of atoms imply Γ- (finitary) universally quantified disjunctions of atoms.

Some Motivation. Craig interpolation has applications in various areas of computer science. Such an area is formal specification theory (see [19, 14]). For structured specifications [3, 29], interpolation ensures a good, compositional, behavior of their semantics [3, 5, 25]. In choosing a logical framework for specifications, one has to find the right balance between expressive power and amenable computational aspects. Therefore, an intermediate choice between the "extremes", full first-order logic and equational logic, might be desirable. We enable (at least partially) such intermediate logics (e.g., the *positive*- or $(\forall\vee)$- logic) as specification frameworks, by showing that they have the interpolation property. Moreover, the

[1] Other results obtained with this technique, including some for second-order and higher-order logic, can be found in the technical report [24].

very general nature of our results w.r.t. signature morphisms sometimes allows one to enrich the class of morphisms used for renaming usually up to arbitrary morphisms, freeing specifications from unnatural constraints, like injectivity of renaming/translation. Some technical details about the applications of our results to formal specifications can be found in Section 5.

Automatic reasoning is another area where interpolation is important and where our results contribute. There, *putting theories together* while still taking advantage, inside their union language, of their available decision procedures [21, 23], relies on interpolation in a crucial way. Moreover, interpolation provides a heuristic to "divide and conquer" a proving task: in order to show $\Gamma_1 \models_{\Sigma_1 \cup \Sigma_2} \Gamma_2$, find some Γ over the syntax $\Sigma_1 \cap \Sigma_2$ and prove the two "simpler" tasks $\Gamma_1 \models_{\Sigma_1} \Gamma$ and $\Gamma \models_{\Sigma_2} \Gamma_2$. For some simpler sub-logics of first-order logic, such as propositional calculus, where there is a finite set of semantically different sentences over any given signature, one can use interpolation also as a disproof technique: if for each $(\Sigma_1 \cap \Sigma_2)$-sentence Γ (there is only a finite number of them) at least one of $\Gamma_1 \models_{\Sigma_1} \Gamma$ or $\Gamma \models_{\Sigma_2} \Gamma_2$ fails, then $\Gamma_1 \models_{\Sigma_1 \cup \Sigma_2} \Gamma_2$ fails. The results of the present paper, although not effectively constructing interpolants, provide information about the existence of interpolants *of a certain type*, helping reducing the space of search. For instance, according to one of the cases of our main result, Theorem 2, the existence of a positive interpolant Γ is ensured by the fact that *either* one of Γ_1 or Γ_2 is positive (lines 2, 3 of table on page 316).

Technical Preliminaries. For simplifying the exposition, set-theoretical foundational issues are ignored in this paper.[2] Given a class $\mathcal{D}$, let $\mathcal{P}(\mathcal{D})$ denote the collection of all subclasses of $\mathcal{D}$. For any $\mathcal{C} \in \mathcal{P}(\mathcal{D})$, let $\overline{\mathcal{C}}$ denote $\mathcal{D} \setminus \mathcal{C}$, that is, the class of all elements in $\mathcal{D}$ which are not in $\mathcal{C}$. Also, given $\mathcal{C}_1, \mathcal{C}_2 \in \mathcal{P}(\mathcal{D})$ let $[\mathcal{C}_1, \mathcal{C}_2]$ denote all classes $\mathcal{C}$ which include $\mathcal{C}_1$ and are included in $\mathcal{C}_2$.

An *operator* on $\mathcal{D}$ is a mapping $F : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$. Let $Id_{\mathcal{D}}$ denote the identity operator. For any operator F on $\mathcal{D}$, let $Fixed(F)$ denote the collection of all *fixed points* of F, that is, $\mathcal{C} \in Fixed(F)$ iff $F(\mathcal{C}) = \mathcal{C}$. An operator F on $\mathcal{D}$ is a *closure operator* iff it is *extensive* ($\mathcal{C} \subseteq F(\mathcal{C})$), *monotone* (if $\mathcal{C}_1 \subseteq \mathcal{C}_2$ then $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$) and *idempotent* ($F(F(\mathcal{C})) = F(\mathcal{C})$).

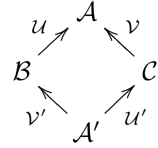
Given a relation $\mathcal{R}$ on $\mathcal{D}$, let $\mathcal{R}$ also denote the operator on $\mathcal{D}$ associated with $\mathcal{R}$, assigning to each $\mathcal{C} \in \mathcal{P}(\mathcal{D})$ the class of all elements from $\mathcal{D}$ in relation with elements in $\mathcal{C}$, that is, $\mathcal{R}(\mathcal{C}) = \{c' \in \mathcal{D} \mid (\exists c \in \mathcal{C})\ c\ \mathcal{R}\ c'\}$. Notice that the operator associated to a reflexive and transitive relation is a closure operator.

Given two classes $\mathcal{C}$ and $\mathcal{D}$ and a mapping $\mathcal{U} : \mathcal{C} \to \mathcal{D}$, we let $\mathcal{U}$ also denote the mapping $\mathcal{U} : \mathcal{P}(\mathcal{C}) \to \mathcal{P}(\mathcal{D})$ defined by $\mathcal{U}(\mathcal{C}') = \{\mathcal{U}(c) \mid c \in \mathcal{C}'\}$ for any $\mathcal{C}' \in \mathcal{P}(\mathcal{C})$. Also, we let $\mathcal{U}^{-1} : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{C})$ denote the mapping defined by $\mathcal{U}^{-1}(\mathcal{D}') = \{c \in \mathcal{C} \mid \mathcal{U}(c) \in \mathcal{D}'\}$ for any $\mathcal{D}' \in \mathcal{P}(\mathcal{D})$. Given two mappings $U, V : \mathcal{P}(\mathcal{C}) \to \mathcal{P}(\mathcal{D})$, we say that U *is included in* V, written $U \sqsubseteq V$, iff $U(\mathcal{C}') \subseteq V(\mathcal{C}')$ for any $\mathcal{C}' \in \mathcal{P}(\mathcal{C})$.

[2] Yet, it is easy to see that references to collections of classes could be easily avoided.

We write the composition of mappings in "diagrammatic order": if $f : A \to B$ and $g : B \to C$ then $f;g$ denotes their composition, regardless of whether f and g are mappings between sets, classes, or collections of classes.

Definition 1. *We say mappings (between classes)* $\mathcal{U}, \mathcal{V}, \mathcal{U}', \mathcal{V}'$ *(see diagram) form a* ***commutative square*** *iff* $\mathcal{V}' ; \mathcal{U} = \mathcal{U}' ; \mathcal{V}$. *A commutative square is a* ***weak amalgamation square*** *iff for any* $b \in \mathcal{B}$ *and* $c \in \mathcal{C}$ *such that* $\mathcal{U}(b) = \mathcal{V}(c)$*, there exists some* $a' \in \mathcal{A}'$ *such that* $\mathcal{V}'(a') = b$ *and* $\mathcal{U}'(a') = c$.



We call this amalgamation square "weak" because a' is not required to be unique.

2 First-Order Logic and Classical Interpolation Revisited

First-Order Logic. A (many-sorted) first-order signature is a triple (S, F, P) consisting of a set S of sort symbols, a set F of function symbols, and a set P of relation symbols. Each function or relation symbol comes with a string of argument sorts, called *arity*, and for function symbols, a result sort. $F_{w \to s}$ denotes the set of function symbols with arity w and result sort s, and P_w the set of relation symbols with arity w. Given a signature Σ, the class of Σ*-models*, $Mod(\Sigma)$ consists of all first-order structures A interpreting each sort symbol s as a non-empty[3] set A_s, each function symbol σ as a function A_σ from the product of the interpretations of the argument sorts to the interpretation of the result sort, and each relation symbol π as a subset A_π of the product of the interpretations of the argument sorts.

The set of Σ*-sentences*, $Sen(\Sigma)$, consists of the usual first-order sentences built from equational and relational atoms by iterative application of logical connectives and quantifiers. The satisfaction of sentences by models ($A \models \gamma$) is the usual Tarskian satisfaction defined inductively on the structure of the sentences. The satisfaction relation can be extended to a relation $\models$ between classes of models $\mathcal{M} \subseteq Mod(\Sigma)$ and sets of sentences $\Gamma \subseteq Sen(\Sigma)$: $\mathcal{M} \models \Gamma$ iff $A \models \gamma$ for all $A \in \mathcal{M}$ and $\gamma \in \Gamma$. This further induces two operators $_^* : \mathcal{P}(Sen(\Sigma)) \to \mathcal{P}(Mod(\Sigma))$ and $_^* : \mathcal{P}(Mod(\Sigma)) \to \mathcal{P}(Sen(\Sigma))$, defined by $\Gamma^* = \{A \mid \{A\} \models \Gamma\}$ and $\mathcal{M}^* = \{\gamma \mid \mathcal{M} \models \{\gamma\}\}$ for each $\Gamma \subseteq Sen(\Sigma)$ and $\mathcal{M} \subseteq Mod(\Sigma)$. The two operators $_^*$ form a Galois connection between $(\mathcal{P}(Sen(\Sigma)), \subseteq)$ and $(\mathcal{P}(Mod(\Sigma)), \subseteq)$. The two composition operators $_^* ; _^*$ are denoted $_^\bullet$ and are called *deduction closure* (the one on sets of sentences) and *axiomatizable hull* (the one on classes of models). We call *elementary classes* the classes of models closed under $_^\bullet$ and *theories* the sets of sentences closed under $_^\bullet$. If $\Gamma, \Gamma' \subseteq Sen(\Sigma)$, we say that Γ *semantically deduces* Γ', written $\Gamma \models \Gamma'$, iff $\Gamma^* \subseteq \Gamma'^*$.

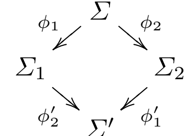
Given two signatures $\Sigma = (S, F, P)$ and $\Sigma' = (S', F', P')$, a signature morphism $\phi : \Sigma \to \Sigma'$ is a triple $(\phi^{st}, \phi^{op}, \phi^{rl})$ mapping the three components in a compatible way. (When there is no danger of confusion, we denote each

[3] Birkhoff-style axiomatizability, which will be used intensively in this paper, depends on the non-emptiness of carriers [27].

of the mappings $\phi^{st}, \phi^{op}, \phi^{rl}$ by ϕ.) Sentence translations rename the sorts, function-, and relation- symbols. For each signature morphism $\phi : \Sigma \to \Sigma'$, the reduct $A'\restriction_\phi$ of a Σ'-model A' is the Σ-model defined by $(A'\restriction_\phi)_\alpha = A'_{\phi(\alpha)}$ for each sort, function, or relation symbol α in the domain signature of ϕ. Let $Mod(\phi) : Mod(\Sigma') \to Mod(\Sigma)$ denote the mapping $A' \mapsto A'\restriction_\phi$. Satisfaction relation has the important property that it is "invariant under change of notation" [17], i.e., for each $\gamma \in Sen(\Sigma)$ and $A' \in Mod(\Sigma')$, $A' \models \phi(\gamma)$ iff $A'\restriction_\phi \models \gamma$.

Interpolation. The original formulation of interpolation [7] is in terms of signature intersections and unions, that is, w.r.t. squares which are pushouts of signature inclusions. However, subsequent advances in modularization theory [3, 13, 14, 9, 4] showed the need of arbitrary pushout squares or even weak amalgamation squares. A general formulation of interpolation is the following:

Definition 2. *Assume a commutative square of signature morphisms (see diagram) and two sets of sentences $\Gamma_1 \subseteq Sen(\Sigma_1)$, $\Gamma_2 \subseteq Sen(\Sigma_2)$ such that $\phi_2'(\Gamma_1) \models_{\Sigma'} \phi_1'(\Gamma_2)$ (i.e., Γ_1 implies Γ_2 on the "union language" Σ'). An* ***interpolant*** *for Γ_1 and Γ_2 is a set $\Gamma \subseteq Sen(\Sigma)$ such that $\Gamma_1 \models_{\Sigma_1} \phi_1(\Gamma)$ and $\phi_2(\Gamma) \models_{\Sigma_2} \Gamma_2$.*



The following two examples show that, without further restrictions on signature morphisms, an interpolant Γ may not be found with *the same type* of sentences as Γ_1 and Γ_2, but with more general ones. In other words, there are sub-first-order logics which do not admit Craig Interpolation within themselves but in a larger (sub-)logic. The first example below shows a square in unconditional equational logic which does not admit unconditional interpolants, but admits a conditional one:

Example 1. Consider the following pushout of algebraic signatures, as in [25]: $\Sigma = (\{s\}, \{d_1, d_2 : s \to s\})$, $\Sigma_1 = (\{s\}, \{d_1, d_2, c : s \to s\})$, $\Sigma_2 = (\{s\}, \{d : s \to s\})$, $\Sigma' = (\{s\}, \{d, c : s \to s\})$, all morphisms mapping the sort s into itself, ϕ_1 and ϕ_2 mapping d_1 and d_2 into themselves and into d, respectively, ϕ_2' mapping d_1 and d_2 into d and c into itself, and ϕ_1' mapping d into itself.

Take $\Gamma_1 = \{(\forall x)d_2(x) = c(d_1(x)),\ (\forall x)d_1(d_2(x)) = c(d_2(x))\}$ and $\Gamma_2 = \{(\forall x)d(d(x)) = d(x)\}$ to be sets of Σ_1-equations and of Σ_2-equations, respectively. It is easy to see that Γ_1 implies Γ_2 in the "union language", i.e., $\phi_2'(\Gamma_1) \models \phi_1'(\Gamma_2)$. But Γ_1 and Γ_2 have no equational Σ-interpolant, because the only equational Σ-consequences of Γ_1 are the trivial ones, of the form $(\forall X)t = t$ with t a Σ-term (since all the nontrivial Σ_1-consequences of Γ_1 contain the symbol c). Yet, Γ_1 and Γ_2 have a conditional-equational interpolant, e.g., $\{(\forall x)d_1(x) = d_2(x) \Rightarrow d_1(x) = d_1(d_1(x))\}$.

The following example shows a situation in which the interpolant cannot even be conditional-equational; it can be a first-order, though:

Example 2. Consider the same pushout of signatures as in previous example and take $\Gamma_1 = \{(\forall x)d_2(x) = d_1(c(x)),\ (\forall x)d_1(d_2(x)) = d_2(c(x))\}$ and

$\Gamma_2 = \{(\forall x)d(d(x)) = d(x)\}$. Again, $\phi_2'(\Gamma_1) \models \phi_1'(\Gamma_2)$. But now Γ_1 and Γ_2 have no conditional-equational Σ-interpolant either, because all nontrivial conditional equations we can infer from Γ_1 contain c (to see this, think in terms of the deduction system for conditional equational logic). Nevertheless, Γ_1 and Γ_2 have a first-order interpolant, e.g., $\{(\forall x)d_1(x) = d_2(x) \Rightarrow (\forall y)d_1(y) = d_1(d_1(y))\}$.

An obstacle to interpolation inside the desired type of sentences in the examples above is the lack of injectivity of ϕ_2 on operation symbols; injectivity on both sorts and operation symbols implies conditional equational interpolation [26].

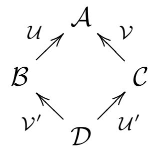
A counterexample given in [4] shows that not even first-order logic admits interpolation without making additional requirements on the square morphisms. We shall shortly prove that for a pushout square to have first-order interpolation, it is sufficient that it has *one* of the morphisms injective *on sorts*. This is, up to our knowledge, the most general known effective criterion for a pushout to have first-order interpolation.

3 Semantic Interpolation

The interpolation problem, despite its syntactic nature, can be regarded *semantically*, on classes of models. Indeed, by the sentence-model duality and the satisfaction condition, we have that:

- $\phi_2'(\Gamma_1) \models \phi_1'(\Gamma_2)$ iff $\phi_2'(\Gamma_1)^* \subseteq \phi_1'(\Gamma_2)^*$ iff $Mod(\phi_2')^{-1}(\Gamma_1^*) \subseteq Mod(\phi_1')^{-1}(\Gamma_2^*)$;
- $\Gamma_1 \models \phi_1(\Gamma)$ iff $\Gamma_1^* \subseteq \phi_1(\Gamma)^*$ iff $\Gamma_1^* \subseteq Mod(\phi_1)^{-1}(\Gamma^*)$;
- $\phi_2(\Gamma) \models \Gamma_2$ iff $\phi_2(\Gamma)^* \subseteq \Gamma_2^*$ iff $Mod(\phi_2)^{-1}(\Gamma^*) \subseteq \Gamma_2^*$.

Therefore, the interpolation property can be restated only in terms of inclusions between classes of models. If Γ is an interpolant of Γ_1 and Γ_2, we will call Γ^* a *semantic interpolant* of Γ_1^* and Γ_2^*. These suggest defining the following broader notion of "semantic interpolation":



Definition 3. *Consider the commutative diagram on the right, together with some $\mathcal{M} \in \mathcal{P}(\mathcal{B})$ and $\mathcal{N} \in \mathcal{P}(\mathcal{C})$ such that $\mathcal{V}'^{-1}(\mathcal{M}) \subseteq \mathcal{U}'^{-1}(\mathcal{N})$. We say that $\mathcal{K} \in \mathcal{P}(\mathcal{A})$ is a* ***semantic interpolant*** *of $\mathcal{M}$ and $\mathcal{N}$ iff $\mathcal{M} \subseteq \mathcal{U}^{-1}(\mathcal{K})$ and $\mathcal{V}^{-1}(\mathcal{K}) \subseteq \mathcal{N}$.*

If we take $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$ to be $Mod(\Sigma)$, $Mod(\Sigma_1)$, $Mod(\Sigma_2)$, $Mod(\Sigma')$ and $\mathcal{U}$, $\mathcal{V}$, $\mathcal{U}'$, $\mathcal{V}'$ to be $Mod(\phi_1)$, $Mod(\phi_2)$, $Mod(\phi_1')$, $Mod(\phi_2')$, respectively, we obtain the concrete first-order case. The connection between semantic interpolation and classical logical interpolation holds only when one considers classes which are *elementary*, i.e., specified by sets of sentences, and the interpolant is also elementary. Rephrasing the interpolation problem semantically allows us to adopt the following "divide and conquer" approach, already sketched in [25]:

1. Find as many semantic interpolants as possible without caring whether they are axiomatizable or not (note that "axiomatizable" will mean "elementary" only within first-order logic, but we shall consider other logics as well);
2. Then, by imposing diverse axiomatizability closures on the two starting classes of models, try to obtain a closed interpolant.

Let $\mathcal{I}(\mathcal{M},\mathcal{N})$ denote the collection of all semantic interpolants of $\mathcal{M}$ and $\mathcal{N}$. The following gives a precise characterization of semantic interpolants together with a general condition under which they exist.

Proposition 1. *Under the hypothesis of Definition 3:*

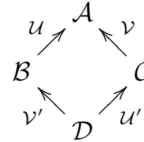
1. $\mathcal{I}(\mathcal{M},\mathcal{N}) = [\mathcal{U}(\mathcal{M}), \overline{\mathcal{V}(\overline{\mathcal{N}})}]$;
2. *If the square is a weak amalgamation square then* $\mathcal{I}(\mathcal{M},\mathcal{N}) \neq \emptyset$.

Definition 4. *Given two classes* $\mathcal{C}$ *and* $\mathcal{D}$, *a mapping* $\mathcal{U} : \mathcal{C} \to \mathcal{D}$ *and a pair of operators* $F = (F_{\mathcal{C}}, F_{\mathcal{D}})$, *we say that* $\mathcal{U}$ ***preserves fixed points of*** F *if* $\mathcal{U}(Fixed(F_{\mathcal{C}})) \subseteq Fixed(F_{\mathcal{D}})$, *that is, for any fixed point of* $F_{\mathcal{C}}$ *we obtain through* $\mathcal{U}$ *a fixed point of* $F_{\mathcal{D}}$; *also we say that* $\mathcal{U}$ ***lifts*** F *if* $F_{\mathcal{D}} ; \mathcal{U}^{-1} \sqsubseteq \mathcal{U}^{-1} ; F_{\mathcal{C}}$, *that is, for any* $\mathcal{D}' \in \mathcal{P}(\mathcal{D})$ *and any* $c \in \mathcal{C}$, *if* $\mathcal{U}(c) \in F_{\mathcal{D}}(\mathcal{D}')$ *then* $c \in F_{\mathcal{C}}(\mathcal{U}^{-1}(\mathcal{D}'))$.

The intuition for the word "lifts" used above comes from the case of the operators $F_{\mathcal{C}}$ and $F_{\mathcal{D}}$ being given by binary relations.

The following theorem is at the heart of all our subsequent results. It gives general criteria under which a weak amalgamation square admits semantic interpolants *closed* under certain generic operators.

Theorem 1. *Consider a weak amalgamation square as in diagram and pairs of operators* $F = (F_{\mathcal{B}}, F_{\mathcal{A}})$ *and* $G = (G_{\mathcal{C}}, G_{\mathcal{A}})$ *such that:*



1. $F_{\mathcal{A}} ; G_{\mathcal{A}} ; F_{\mathcal{A}} = F_{\mathcal{A}} ; G_{\mathcal{A}}$;
2. $G_{\mathcal{C}}$ *and* $G_{\mathcal{A}}$ *are closure operators;*
3. $\mathcal{U}$ *preserves fixed points of* F;
4. $\mathcal{V}$ *lifts* G.

Then for each $\mathcal{M} \in Fixed(F_{\mathcal{B}})$ *and* $\mathcal{N} \in Fixed(G_{\mathcal{C}})$ *such that* $\mathcal{V}'^{-1}(\mathcal{M}) \subseteq \mathcal{U}'^{-1}(\mathcal{N})$, $\mathcal{M}$ *and* $\mathcal{N}$ *have a semantic interpolant* $\mathcal{K}$ *in* $Fixed(F_{\mathcal{A}}) \cap Fixed(G_{\mathcal{A}})$.

The operators above will be conveniently chosen in the next section to be closure operators characterizing axiomatizable classes of models. The two types of axiomatizability that we consider as attached to F and G need not be the same, i.e., the classes $\mathcal{M}$ and $\mathcal{N}$ need not be axiomatizable by the same type of first-order sentences. And in the most fortunate cases, as we shall see below, the interpolant is able to capture and even strengthen the properties of both classes.

4 New Interpolation Results

We next give a series of interpolation results for various types of first-order sentences.[4] Our semantic approach exploits axiomatizability results; since these results use model (homo)morphisms, we first briefly recall some definitions.

Given two Σ-models A and B, a *morphism* $h : A \to B$ is an S-sorted function $(h_s : A_s \to B_s)_{s \in S}$ that commutes with operations and preserves relations.

[4] See the technical report [24] for more interpolation results.

Models and model morphisms form a category denoted $Mod(\Sigma)$ too (just like the class of models), with composition defined as sort-wise function composition. For each signature morphism ϕ, the mapping $Mod(\phi)$ can be naturally extended to a functor. A *surjective* (*injective*) morphism is a morphism which is surjective (injective) on each sort. Because of the weak form of commutation imposed on morphisms w.r.t. the relational part of models, relations and functions do not behave similarly along arbitrary morphisms, but only along closed ones: a morphism $h : A \to B$ is called *closed* if the relation preservation condition holds in the "iff" form, that is, for each predicate symbol π, $(a_1, \ldots, a_n) \in A_\pi$ iff $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in B_\pi$. A morphism $h : A \to B$ is called *strong* if the target relations are covered through h by the source relation, that is, for each predicate symbol π and $(b_1, \ldots, b_n) \in B_\pi$, there exists $(a_1, \ldots, a_n) \in A_\pi$ such that $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) = (b_1, \ldots, b_n)$. Closed injective morphisms and strong surjective morphisms naturally capture the notions of embedding and homomorphic image respectively.

We next define some types of first-order sentences.

- $\mathcal{FO}$: first-order sentences;
- $\mathcal{P}os$: *positive* sentences, that is, constructed inductively from atomic formulae by means of any first-order constructs, except negation;
- $\forall$: sentences $(\forall x_1, x_2, \ldots, x_k)e$, where e is a quantifier free formula;
- $\exists$: sentences $(\exists x_1, x_2, \ldots, x_k)e$, where e is a quantifier free formula;
- $\mathcal{UH}$, *universal Horn clauses*, that is, $(\forall x_1, x_2, \ldots, x_k)(e_1 \wedge e_2 \ldots \wedge e_p) \Rightarrow e$, with e_i, e atomic formulae;
- $\mathcal{UA}$, *universal atoms*, that is, $(\forall x_1, x_2, \ldots x_k)e$, where e is an atomic formula;
- $\forall\vee$, universally quantified disjunctions of atoms, i.e., $(\forall x_1, x_2, \ldots, x_k)(e_1 \vee e_2 \ldots \vee e_p)$ where e_i are atomic formulae;
- $\mathcal{FO}_\infty, \mathcal{UH}_\infty, \forall\vee_\infty$, the infinitary extensions of $\mathcal{FO}, \mathcal{UH}, \forall\vee$, respectively; in the former case, infinite conjunction and disjunction is allowed; in the latter two cases, $e_1 \wedge e_2 \ldots \wedge e_p$ and $e_1 \vee e_2 \ldots \vee e_p$ are replaced by any possibly infinite sentence- conjunction and disjunction respectively.

We assume the reader familiar with some basic model theoretic notions, such as submodels, products, filtered products, ultraproducts, ultrapowers and ultraradicals (see, e.g., [6, 20]). Recall though that, given a family $(A_i)_{i\in I}$ of models and a filter $\mathcal{F}$ on I (i.e., a Boolean filter $\mathcal{F} \subseteq 2^I$), the *filtered product* of $(A_i)_{i\in I}$ over $\mathcal{F}$, denoted $\prod_{\mathcal{F}} A_i$, has the carrier $B = \prod_{i\in I} A_i/{\equiv}$, where $\equiv$ is given by $(a_i)_{i\in I} \equiv (b_i)_{i\in I}$ iff $\{i \in I \mid a_i = b_i\} \in \mathcal{F}$. Operations are defined by $B_\sigma((a_i^1)_{i\in I}/{\equiv}, \ldots, (a_i^n)_{i\in I}/{\equiv}) = (B_\sigma(a_i^1, \ldots, a_i^n))_{i\in I}/{\equiv}$ and $B_\pi = \{((a_i^1)_{i\in I}/{\equiv}\ , \ldots, (a_i^n)_{i\in I}/{\equiv}) \mid \{i \in I \mid (a_i^1, \ldots, a_i^n) \in A_{i\pi}\} \in \mathcal{F}\}$. If $\mathcal{F}$ is an *ultrafilter*, then $\prod_{\mathcal{F}} A_i$ is said to be an *ultraproduct*; moreover, if all A_i's are equal to some model A, then $\prod_{\mathcal{F}} A_i$ is written $A^I/_{\mathcal{F}}$ and said to be the *ultrapower* of A over $\mathcal{F}$; in this latter case, A is said to be an *ultraradical* of $A^I/_{\mathcal{F}}$.
Consider the following binary relations on Σ-models:

- A ***S*** B iff B is isomorphic to a *submodel* of A;
- A ***Ext*** B iff B is isomorphic to an *extension* of A, i.e., to a model C such that A is a submodel of C;

- A ***H*** B iff there exists a *surjective* morphism between A and B;
- A ***Hs*** B iff there exists a *strong surjective* morphism between A and B;
- A ***Ur*** B iff B is an *ultraradical* of a model isomorphic to A, i.e., if A is isomorphic to an *ultrapower* of B

Recall that any binary relation, in particular the ones on $Mod(\Sigma)$ above, has an associated operator bearing the same name. Besides these operators, we shall also consider the operators P, Fp, and Up on $Mod(\Sigma)$ defined below:

- $P(\mathcal{M}) = \mathcal{M} \cup \{$all *products* of models in $\mathcal{M}\}$;
- $Fp(\mathcal{M}) = \mathcal{M} \cup \{$all *filtered products* of models in $\mathcal{M}\}$;
- $Up(\mathcal{M}) = \mathcal{M} \cup \{$all *ultraproducts* of models in $\mathcal{M}\}$;

All these constructions are considered up to isomorphism; for instance, the operator Up "grabs" into the class not only the ultraproducts standardly constructed as quotients of direct products, but all models isomorphic to them.

The next proposition collects some known axiomatizability results. For details, the reader is referred to [6] (Section 5.2), [20] (Sections 25 and 26), [1], [22], and [9]. Below, e.g., the pair $(\mathcal{UA}, \{S, H, P\})$ corresponds to the famous Birkhoff Theorem (a class of algebras is equationally axiomatizable iff it is closed under subalgebras, homomorphic images, and products) and the pair $(\mathcal{FO}, \{Up, Ur\})$ corresponds to the Keisler-Shelah Theorem (a class of first-order models is elementary iff it is closed under ultrapowers and ultraradicals).

Proposition 2. *If the pair (T, Ops), consisting of a type T of Σ-sentences and a set Ops of operators on $Mod(\Sigma)$, is one of $(\mathcal{FO}, \{Up, Ur\})$, $(\mathcal{P}os, \{Up, Ur, H\})$, $(\forall, \{S, Up\})$, $(\exists, \{Ext, Up, Ur\})$, $(\mathcal{UH}, \{S, Fp\})$, $(\mathcal{UA}, \{S, H, P\})$, $(\forall\vee, \{Hs, S, Up\})$, $(\mathcal{UH}_\infty, \{S, P\})$, $(\forall\vee_\infty, \{Hs, S\})$, then $\mathcal{M} \subseteq Sen(\Sigma)$ is of the form Γ^* with $\Gamma \subseteq T$ iff $\mathcal{M}$ is a fixed point of all the operators in Ops.*

Consider the following "syntactic" properties for a morphism $\phi : \Sigma \to \Sigma'$, where $\Sigma = (S, F, P)$ and $\Sigma' = (S', F', P')$:

- (IS) ϕ is ***injective on sorts***;
- (IR) ϕ is ***injective on relation symbols***;
- (I) ϕ is ***injective*** on sorts, operation- and relation- symbols
- (RS) there are no operation symbols in $F' \setminus \phi(F)$, having the ***result sort*** in $\phi(S)$.

Proposition 3. *For each signature morphism $\phi : \Sigma \to \Sigma'$,*

1. *$Mod(\phi)$ preserves fixed points of P, Fp, Up;*
2. (I) $\Rightarrow$ *$Mod(\phi)$ lifts S, H, Hs and preserves fixed points of Ext* [9].
3. (IS) *and* (RS) $\Rightarrow$ *$Mod(\phi)$ preserves fixed points of S, Hs, and lifts Ext;*
4. (IS), (IR) *and* (RS) $\Rightarrow$ *$Mod(\phi)$ preserves fixed points of H;*
5. (IS) $\Rightarrow$ *$Mod(\phi)$ lifts Ur;*

The table below states interpolation results for diverse types of sentences. It should be read as: given a weak amalgamation square of signatures as in Definition 2 and $\Gamma_1 \subseteq Mod(\Sigma_1)$, $\Gamma_2 \subseteq Mod(\Sigma_2)$, if Γ_1 and Γ_2 are sentences of the indicated types such that $\phi'_2(\Gamma_1) \models \phi'_1(\Gamma_2)$, then there exists an interpolant Γ of the indicated type; the semantic conditions under which this situation holds are given in the $Mod(\phi_1)$- and $Mod(\phi_2)$- columns of the table, with the meaning that $Mod(\phi_1)$ *preserves* fixed points of the indicated operator and $Mod(\phi_2)$ *lifts* the indicated operator. (*Id* is the identity operator.) These semantic conditions are implied by the syntactic conditions listed in the ϕ_1- and ϕ_2- columns; "any" means that no restriction is posed on the signature morphism.

	Γ_1 Type	Γ_2 Type	Γ Type	$Mod(\phi_1)$ preserves	$Mod(\phi_2)$ lifts	ϕ_1	ϕ_2
1	$\mathcal{FO}$	$\mathcal{FO}$	$\mathcal{FO}$	*Up*	*Ur*	any	(IS)
2	$\mathcal{FO}$	$\mathcal{P}os$	$\mathcal{P}os$	*Up*	*H* ; *Ur*	any	(I)
3	$\mathcal{P}os$	$\mathcal{FO}$	$\mathcal{P}os$	*Up* ; *H*	*Ur*	(IS), (IR), (RS)	(IS)
4	$\mathcal{FO}$	$\forall$	$\forall$	*Up*	*S*	any	(I)
5	$\forall$	$\mathcal{FO}$	$\forall$	*Up* ; *S*	*Id*	(IS), (RS)	any
6	$\forall$	$\mathcal{P}os$	$\forall\vee$	*Up* ; *S*	*Hs*	(IS), (RS)	(I)
7	$\mathcal{FO}$	$\exists$	$\exists$	*Up*	*Ext*; *Ur*	any	(IS), (RS)
8	$\exists$	$\mathcal{FO}$	$\exists$	*Up* ; *Ext*	*Ur*	(I)	(IS)
9	$\mathcal{FO}$	$\mathcal{UH}$	$\mathcal{UH}$	*Fp*	*S*	any	(I)
10	$\mathcal{UH}$	$\mathcal{FO}$	$\mathcal{UH}$	*Fp* ; *S*	*Id*	(IS), (RS)	any
11	$\mathcal{UH}$	$\mathcal{UA}$	$\mathcal{UA}$	*P*	*S* ; *H*	any	(I)
12	$\mathcal{UA}$	$\mathcal{FO}$	$\mathcal{UA}$	*P* ; *S* ; *H*	*Id*	(IS), (IR), (RS)	any
13	$\mathcal{UH}$	$\mathcal{P}os$	$\mathcal{UA}$	*P* ; *S*	*H*	(IS),(RS)	(I)
14	$\mathcal{FO}$	$\forall\vee$	$\forall\vee$	*Up*	*S* ; *Hs*	any	(I)
15	$\forall\vee$	$\mathcal{FO}$	$\forall\vee$	*Up* ; *S* ; *Hs*	*Id*	(IS), (RS)	any
16	$\mathcal{UH}_\infty$	$\mathcal{UA}$	$\mathcal{UA}$	*P*	*S* ; *H*	any	(I)
17	$\mathcal{UH}_\infty$	$\mathcal{FO}_\infty$	$\mathcal{UH}_\infty$	*P* ; *S*	*Id*	(IS), (RS)	any
18	$\mathcal{FO}_\infty$	$\forall\vee_\infty$	$\forall\vee_\infty$	*Id*	*S* ; *Hs*	any	(I)
19	$\forall\vee_\infty$	$\mathcal{FO}_\infty$	$\forall\vee_\infty$	*S* ; *Hs*	*Id*	(IS), (RS)	any
20	$\mathcal{FO}$	$\forall\vee_\infty$	$\forall\vee$	*Up*	*S* ; *Hs*	any	(I)

Theorem 2. *The results stated in this table hold, i.e., in each of the 20 cases, if ϕ_1 and ϕ_2 satisfy the indicated properties, Γ_1 and Γ_2 have the indicated types and $\phi'_2(\Gamma_1) \models \phi'_1(\Gamma_2)$, then there exists an interpolant Γ of the indicated type.*

Let us discuss the results listed in the table above. The syntactic conditions on signature morphisms are in many cases weaker than, or equal to, injectivity (I). In fact, if we consider only relational languages, i.e., without operation symbols, *all* the conditions are so (because (RS) becomes vacuous). As for operation symbols, it is interesting to note that (RS) comprises the principle of data encapsulation expressed in algebraic terms [16]. As also suggested by the examples in Section 2, it seems that the degree of generality that one can allow on signature morphism

increases with the expressive power of a logic. For instance, line 1 says that first-order interpolation holds whenever the righthand morphism is injective on sorts (and, in fact, since in full first-order logic Craig interpolation is equivalent to the symmetrical property of Robinson consistency,[5] *either one* of the morphisms being injective on sorts would do). On the other hand, universal Horn clauses (lines 9 and 10), and then universal atoms (lines 11, 12, 13) require stronger and stronger assumptions on the signature morphisms. Our results say more than interpolation within a certain type T of sentences: the interpolant has type T provided *one* of the starting sets has type T. Particularly interesting results are listed in lines 6, 13, and 20, where the interpolant strictly "improves" the type of both sides. Regarding finiteness of Γ, as noted in [9], it is easy to see that if Γ_2 is finite, by compactness of first-order logic, Γ can be chosen to be also finite in our cases of finitary sub-first-order logics. On the other hand, the finiteness of Γ_1 does not necessarily imply the finite axiomatizability of Γ^* (this follows by a famous theorem due to Kleene).

5 Applications to Formal Specification

Craig interpolation is a an important/desired property in many areas. Next we consider some applications of our interpolation results to formal specification and module algebra.

In formalisms for modularization [3, 13, 29], modules are built by composing other modules via specific operations. One typically starts with *flat* (or *basic*) modules, which are pairs (Σ, Γ) comprising a signature Σ and a set of Σ-sentences Γ. According to [3], one of the most natural semantics of modules, also called *flat semantics*, is given by their corresponding theories; for example the semantics of a basic module (Σ, Γ) is the theory $(\Sigma, \Gamma^\bullet)$. Diverse operations are used to build up structured theories, among which the *export* (or *information hiding*) and *combination* (or *sum*) operators [3] (or [13]), $\Box$ and $+$. $\Box$ restricts the interface of the theory (Σ, Γ) to common symbols of Σ' and Σ, while $+$ just puts together two theories in their union signature. Formally, for each signature Σ' and theory (Σ, Γ), let $\Sigma' \Box (\Sigma, \Gamma)$ be $(\Sigma' \cap \Sigma, \iota^{-1}(\Gamma))$, where $\iota : \Sigma' \cap \Sigma \hookrightarrow \Sigma$; and for theories (Σ_1, Γ_1) and (Σ_2, Γ_2), let $(\Sigma_1, \Gamma_1) + (\Sigma_2, \Gamma_2)$ be $(\Sigma_1 \cup \Sigma_2, (\Gamma_1 \cup \Gamma_2)^\bullet)$. A very desirable property of specification frameworks is the following *restricted distributivity law*:

$$\Sigma' \Box ((\Sigma_1, \Gamma_1) + (\Sigma_2, \emptyset^\bullet)) = (\Sigma' \Box (\Sigma_1, \Gamma_1)) + (\Sigma' \Box (\Sigma_2, \emptyset^\bullet))$$

As discussed in [3, 13], full distributivity does not typically hold. It is shown in [3] that, in first-order logic, restricted distributivity is implied by interpolation. Their proof is rather logic-independent, so it works for any logic that has first-order signatures and satisfies interpolation. In particular, it works for all the sublogics of (finitary or infinitary) first-order logic appearing in the table that precedes Theorem 2. Thus our interpolation results show that the restricted

[5] This is not true however for our examples of sub-first-order logics.

distributivity law holds in module algebra developed within many logical frameworks intermediate between full first-order logic and equational logic.

Another application to formal specifications relies on the fact that interpolation entails a *compositional behavior* of the semantics of structured specifications, by ensuring that the two alternative semantics, the flat and the structured ones, coincide. There are good reasons to *not* always consider the flat semantics of module expressions, but rather to *keep the structure* of modules [29, 5, 18]. In the case of hiding, $\Sigma'\Box(\Sigma, \Gamma)$ provides more information than $(\Sigma', \Gamma^\bullet \cap Sen(\Sigma'))$: (1) Γ might be finite, showing that $\Gamma^\bullet$, maybe unlike $\Gamma^\bullet \cap Sen(\Sigma')$, is finitely presented; (2) while the theory of all Σ'-reducts of (Σ, Γ) (i.e., all visible parts of the possible implementations of the theory) is indeed $\Gamma^\bullet \cap Sen(\Sigma')$, usually not any model of $\Gamma^\bullet \cap Sen(\Sigma')$ is a Σ-reduct of a model of (Σ, Γ); hence the theory does not describe precisely the intended semantics on classes of models.

To understand the role played by interpolation, consider the situation when a module $\Sigma'\Box(\Sigma, \Gamma)$ is imported and its interface (Σ') renamed via a signature morphism $j : \Sigma' \to \Sigma''$ in the importing context. The flat semantics of the renamed module is $(\Sigma'', j(\Gamma^\bullet \cap Sen(\Sigma'))^\bullet)$. On the other hand, the renamed module itself might be regarded construcively as an information hiding module whose interface is Σ'' and whose base module is a consistent renaming of (Σ, Γ). This is achieved by taking the pushout $(\Sigma'' \hookrightarrow \Sigma_0, j_0 : \Sigma \to \Sigma_0)$ of $(\Sigma' \hookrightarrow \Sigma, j : \Sigma' \to \Sigma'')$, yielding the new module $\Sigma''\Box(\Sigma_0, j_0(\Gamma))$. One can show *using interpolation* that the modular and the flat semantics are equivalent, that is, $j(\Gamma^\bullet \cap Sen(\Sigma'))^\bullet = j_0(\Gamma)^\bullet \cap Sen(\Sigma'')$. This desirable semantical equivalence is shown by our results to hold for several first-order sublogics. More precisely, lines 3,5,15 in the table preceding Theorem 2 show that the framework may be restricted to positive-, universal-, or [universal quantification of atom disjunction]- logics. Moreover, line 19 shows the same thing for the [universal quantification of possibly infinite atom disjunction]-logic. According to these results, the renaming morphism j can be allowed to be injective *on sorts* in the case of positive logic and *any* morphism in the other three cases. Note that lines 2,4,14,18 list results complementary to the above, and generalize those in [9]. These latter results relax the requirements not on the renaming morphism, but on the *hiding morphism* (allowing one to replace the inclusion $\Sigma' \hookrightarrow \Sigma$ with an arbitrary signature morphism).

Within a specification framework, one should not commit herself to a particular kind of first-order sub-logic, but rather use the available power of expression on a by-need basis, keeping flexible the border between expressive power and effective/efficient decision or computation. The issue of coexistence of different logical systems brings up a third application of our results. The various logical systems that one would like to use should not be simply "swallowed" by a richer universal logic that encompasses them all, but rather integrated using logic translations. This methodology, which is the meta-logical counterpart of keeping structured (i.e., unflattened) the specifications themselves, is followed for instance in CafeOBJ [11, 12]. The underlying logical structure of this system can be formalized as a *Grothendieck institution* [8], which provides a means of

building specifications inside the minimal needed logical system. The framework is initially presented as an *indexed institution*, i.e., a family of logical systems with translations between them, and then flattened by a Grothendieck construction.

Lifting interpolation from the component institutions to the Grothendieck institution was studied in [10]; a criterion is given there for lifting interpolation, consisting mainly of three conditions: (1) that the component institutions have interpolation (for some designated pushouts of signatures); (2) that the involved institution comorphisms have interpolation; (3) that each pullback in the index category yields an interpolating square of comorphisms. We give just one example showing that, via the above conditions, some of our interpolation results can be used for putting together in a consistent way two very interesting logical systems: (finitary) first-order logic ($\mathcal{FO}$) and the logic of universally quantified possibly infinite disjunctions of atoms ($\forall\vee_\infty$). While the former is a well-established logic, the latter has the ability of expressing some important properties, not expressible in the former, such as *accessibility* of models, e.g., $(\forall x)(x = 0 \vee x = s(0) \vee x = s(s(0)) \vee \ldots)$ for natural numbers. If one combines the expressive power of these two logics, initiality conditions are also available, e.g., the above accessibility condition ("no junk") can be complemented with the "no confusion" statement $\neg \bigvee_{i,j\in\mathbb{N},i<j} s^i(0) = s^j(0)$. Since the two logical systems have the same signatures, condition (2) above is trivially satisfied. Moreover, our results stated in lines 1 and 19 of the table preceding Theorem 2 ensure condition (1) for some very wide class of signature pushouts. Finally, condition (3) is fulfilled by the result in line 20, which states that formulae from the two logics have interpolants *in their intersection logic*, that of universally quantified (finite) conjunctions of atoms.

6 Related Work and Concluding Remarks

The idea of using axiomatizability properties for proving Craig interpolation first appeared, up to our knowledge, in [27] in the case of many-sorted equational logic. Then [25] generalized this to an arbitrary pullback of categories, by considering some Birkhoff-like operators on those categories, with results applicable to different versions of equational logic. An institution-independent relationship between Birkhoff-like axiomatizability and Craig interpolation was depicted in [9], using a concept of *Birkhoff institution*. If we disregard combination of logics and flatten to the least logic, the results in lines 2,4,14,18 of the table preceding Theorem 2 can be also found in [9]. Our Theorem 1 generalizes the previous "semantic" results, bringing the technique of semantic interpolation, we might say, up to its limit. The merit of Theorem 1 is that it provides general conditions under which a semantic interpolant has a syntactic counterpart (i.e., it is axiomatizable). This theorem solves only half of the interpolation problem; concrete lifting and preserving conditions, as well as certain inclusions between operators, still have to be proved. Thus, in this paper, we provide a general methodology for proving interpolation results. We have also followed this methodology working

out many concrete examples. The list of sub-first-order-logics that fit our framework is of course open for other suitably axiomatizable logics; and so are the possible combinations between these logics, which might guarantee interpolants even simpler than the types of formulae of *both* logics, as shown by some of our results. Regarding our combined interpolation results, it is worth pointing out that they are not overlapped with, but rather complementary to, the ones in [10] for Grothendieck institutions. There, some combined interpolation properties are previously assumed, in order to ensure interpolation in the resulted larger logical system.

An interesting fact to investigate would be to which extent can syntactically-obtained interpolation results "compete" with our semantic results. While it is true that the syntactic proofs are usually constructive, they do not seem to provide information on the type of the interpolant comparable to what we gave here. In particular, since the diverse Gentzen systems for first-order logic with equality have only partial cut elimination [15], an appeal to the non-equality version of the language, by adding appropriate axioms for equality in the theory, is needed; moreover, dealing with function symbols requires a further appeal to an encoding of functions as relations, again with the cost of adding some axioms. All these transformations make even some presumably very careful syntactic proofs rather indirect and obliterating, and sometimes place the interpolant way outside the given subtheory - this is the reason why an interpolation theorem for equational logic was not known until a separate, specific proof was given in [28]. Yet, comparing and paralleling (present or future) semantic and syntactic proofs seems fruitful for deepening our understanding of Craig interpolation, this extremely complex and resourceful, purely syntactic and yet surprisingly semantic, property of logical systems.

References

1. H. Andréka and I. Németi. A general axiomatizability theorem formulated in terms of cone-injective subcategories. In *Universal Algebra*, volume 29, pages 13–35. 1982.
2. J. Barwise and J. Feferman. *Model-Theoretic Logics*. Springer, 1985.
3. J. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372, 1990.
4. T. Borzyszkowski. Generalized interpolation in CASL. *Inf. Process. Lett.*, 76(1-2):19–24, 2000.
5. T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*, 286(2):197–245, 2002.
6. C. C. Chang and H. J. Keisler. *Model Theory*. North Holland, Amsterdam, 1973.
7. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen Theorem. *Journal of Symbolic Logic*, 22:250–268, 1957.
8. R. Diaconescu. Grothendieck institutions. *Appl. Categorical Struct.*, 10(4):383–402, 2002.
9. R. Diaconescu. An institution-independent proof of Craig interpolation theorem. *Studia Logica*, 77(1):59–79, 2004.
10. R. Diaconescu. Interpolation in Grothendieck institutions. *Theoretical Computer Science*, 311:439–461, 2004.

11. R. Diaconescu and K. Futatsugi. *CafeOBJ Report*. World Scientific, 1998. AMAST Series in Computing, volume 6.
12. R. Diaconescu and K. Futatsugi. Logical foundations of CafeOBJ. *Theoretical Computer Science*, 285:289–318, 2002.
13. R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularization. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge, 1993.
14. T. Dimitrakos and T. Maibaum. On a generalized modularization theorem. *Information Processing Letters*, 74(1–2):65–71, 2000.
15. J. H. Gallier. *Logic for computer science. Foundations of automatic theorem proving*. Harper & Row, 1986.
16. J. Goguen. Types as theories. In *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.
17. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, January 1992.
18. J. Goguen and G. Roşu. Composing hidden information modules over inclusive institutions. In *From Object Orientation to Formal Methods: Dedicated to the memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 96–123. Springer, 2004.
19. D. G. J. Bicarregui, T. Dimitrakos and T. Maibaum. Interpolation in practical formal development. *Logic Journal of the IGPL*, 9(1):231–243, 2001.
20. J. D. Monk. *Mathematical Logic*. Springer-Verlag, 1976.
21. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
22. I. Németi and I. Sain. Cone-implicational subcategories and some Birkhoff-type theorems. In *Universal Algebra*, volume 29, pages 535–578. 1982.
23. D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12:291–302, 1980.
24. A. Popescu, T. Şerbănuţă, and G. Roşu. A semantic approach to interpolation. Technical Report UIUCDCS-R-2005-2643, University of Illinois at Urbana-Champaign.
25. G. Roşu and J. Goguen. On equational Craig interpolation. *Journal of Universal Computer Science*, 6:194–200, 2000.
26. P. H. Rodenburg. Interpolation in conditional equational logic. *Fundam. Inform.*, 15(1):80–85, 1991.
27. P. H. Rodenburg. A simple algebraic proof of the equational interpolation theorem. *Algebra Universalis*, 28:48–51, 1991.
28. P. H. Rodenburg and R. van Glabbeek. An interpolation theorem in equational logic. Technical Report CS-R8838, CWI, 1988.
29. D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988.

First-Order and Counting Theories of ω-Automatic Structures

Dietrich Kuske[1] and Markus Lohrey[2]

[1] Institut für Informatik, Universität Leipzig, Germany
[2] Universität Stuttgart, FMI, Germany
kuske@informatik.uni-leipzig.de, lohrey@informatik.uni-stuttgart.de

Abstract. The logic $\mathcal{L}(\mathcal{Q}_u)$ extends first-order logic by a generalized form of counting quantifiers ("the number of elements satisfying ... belongs to the set C"). This logic is investigated for structures with an injective ω-automatic presentation. If first-order logic is extended by an infinity-quantifier, the resulting theory of any such structure is known to be decidable [4]. It is shown that, as in the case of automatic structures [13], also modulo-counting quantifiers as well as infinite cardinality quantifiers ("there are $\varkappa$ many elements satisfying ...") lead to decidable theories. For a structure of bounded degree with injective ω-automatic presentation, the fragment of $\mathcal{L}(\mathcal{Q}_u)$ that contains only effective quantifiers is shown to be decidable and an elementary algorithm for this decision is presented. Both assumptions (ω-automaticity and bounded degree) are necessary for this result to hold.

1 Introduction

Automatic structures were introduced in [8, 11]. The idea goes back to the concept of automatic groups [6]. Roughly speaking, a structure is called automatic if the elements of the universe are represented (not necessarily uniquely) as words from a regular language and every relation (including the identity) of the structure can be recognized by a finite state automaton with several heads that proceed synchronously. Automatic structures received increasing interest during the last years [1, 4, 9, 12, 14, 16]. Recently, automatic structures were generalized to ω-automatic structures by the use of Büchi-automata instead of automata on finite words [4]. One of the main motivations for investigating (ω-)automatic structures is the fact that every (ω-)automatic structure has a decidable first-order theory [4, 11]. For automatic structures, this result has been extended to first-order logic with modulo quantifiers [13] and the quantifier "there exist infinitely many" (infinity quantifier) [4]. The infinity quantifier was also shown to lead to decidable theories in the realm of ω-automatic structures [3, 4] with injective presentations (i.e., if the elements of the structure are represented by unique ω-words).[1] While there exist automatic structures with a non-elementary

[1] The decidability proof of [4, Thm. 2.1] assumes an injective ω-automatic presentation. [4, Prop. 5.2] states that any ω-automatic structure has such an injective presentation, but the proof is spurious (cf. Remark 2.1). So we safely use the decidability for injective presentations, only.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 322–336, 2006.
© Springer-Verlag Berlin Heidelberg 2006

first-order theory [4], the first-order theory of any automatic structure of bounded degree is elementarily decidable; more precisely, an upper bound of triply exponential alternating time with a linear number of alternations was shown in [16].

The overall theme of this paper is to extend these results from automatic structures to ω-automatic structures and to consider more involved logics. In a *first step*, we extend first-order logic by modulo-counting quantifiers as in [13] and exact counting quantifiers for infinite cardinals. We show that any injectively ω-automatic structure has a decidable theory in this logic (Theorem 2.8). This extends [13, Theorem 3.2] from automatic to injectively ω-automatic structures and [4, Theorem 2.1] from first-order logic with an infinity quantifier to a further extension of this logic. The proof is based on automata-theoretic constructions, in particular an analysis of successful runs in Muller automata.

In a *second step*, we consider an even more powerful logic that we call $\mathcal{L}(\mathcal{Q}_u)$, which is a finitary fragment of the logic $\mathcal{L}_{\infty,\omega}(\mathcal{Q}_u)^\omega$ from [10]. In this logic $\mathcal{L}(\mathcal{Q}_u)$ one may use generalized quantifiers of the form $\mathcal{Q}_{\mathcal{C}} y : (\psi_1(y), \ldots, \psi_n(y))$, where y is a first-order variable and $\mathcal{C}$ is an n-ary relation on cardinals. To determine the truth of this formula in a model $\mathcal{A}$, one first determines the cardinalities of the sets defined by the formulas $\psi_i(y)$ $(1 \leq i \leq n)$. If the tuple of these cardinalities belongs to the relation $\mathcal{C}$, then the formula is true. All quantifiers mentioned so far are special instances of these generalized quantifiers. But, e.g., also the Härtig quantifier ("there are as many ... as ...") falls into this category.

For every fragment $\mathcal{L}$ of $\mathcal{L}(\mathcal{Q}_u)$ that contains only countably many generalized quantifiers, and every injectively ω-automatic structure $\mathcal{A}$ of bounded degree, we prove that the $\mathcal{L}$-theory of $\mathcal{A}$ can be decided by a Turing-machine with oracle access to the relations $\mathcal{C}$ that are allowed in the fragment $\mathcal{L}$. Moreover, this Turing-machine works in triply exponential space (Theorem 3.7). This extends [16, Theorem 3] since it applies to (1) injectively ω-automatic structures as opposed to automatic structures and (2) to first-order logic extended by generalized quantifiers. This second main result rests on [10] where Hanf-locality is shown for the logic $\mathcal{L}(\mathcal{Q}_u)$. Our algorithm therefore has to determine how often a given neighborhood is realized (up to isomorphism) in the structure. Differently, in the proof of [16, Theorem 3] a similar locality principle is used to effectively bound the search space of quantifiers to short words.

From Theorem 3.7 we deduce that every $\mathcal{L}$-definable relation over an injectively ω-automatic structure of bounded degree is *effectively* first-order definable and therefore *effectively* regular (Corollary 3.9). If effectiveness is not demanded, first-order definability can be easily deduced also for non-ω-automatic structures of bounded degree from [10].

Note that our results require a structure to be ω-automatic and of bounded degree. We finish the technical part of the paper by showing that both these assumptions are necessary, namely that our results do not hold for recursive structures of bounded degree, nor for locally finite automatic (and hence locally finite injectively ω-automatic) structures.

Proofs that are omitted due to space restrictions can be found in the technical report [15].

2 ω-Automatic Structures, Infinity and Modulo Quantifiers

2.1 Definitions and Known Results

This section introduces automata on finite and on infinite words, (ω-)automatic structures, and logics, and recalls some basic results concerning these concepts. For more details, see [17, 18] for automata theoretic issues, [4, 11, 13] for ω-automatic structures, and [7] as far as logics are concerned.

Büchi-automata. Let Γ be a finite alphabet. With Γ^* we denote the set of all finite words over the alphabet Γ. The set of all nonempty finite words is Γ^+. An *ω-word* over Γ is an infinite ω-sequence $w = a_0 a_1 a_2 \cdots$ with $a_i \in \Gamma$, we set $w(i) = a_i$ for $i \in \mathbb{N}$. A (nondeterministic) *Büchi-automaton* M is a tuple $M = (Q, \Gamma, \delta, \iota, F)$, where Q is a finite set of states, $\iota \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta \subseteq Q \times \Gamma \times Q$ is the transition relation. If $\Gamma = \Sigma^n$ for some alphabet Σ, then we speak of an *n-dimensional Büchi-automaton over Σ*. A *run* of M on an ω-word $w = a_0 a_1 a_2 \cdots$ is an ω-word $r = p_0 p_1 p_2 \cdots$ over the set of states Q such that $(p_i, a_i, p_{i+1}) \in \delta$ for all $i \geq 0$. The run r is *successful* if $p_0 = \iota$ and there exists a final state from F that occurs infinitely often in r. The language $L_\omega(M) \subseteq \Gamma^\omega$ defined by M is the set of all ω-words for which there exists a successful run. An ω-language $L \subseteq \Gamma^\omega$ is *regular* if there exists a Büchi-automaton M with $L_\omega(M) = L$. The class of all regular ω-languages is closed under boolean operations and projections [17]. For two Büchi-automata M_1 and M_2 with n_1 and n_2 many states, resp., there exists a Büchi-automaton with $3 \cdot n_1 \cdot n_2$ many states accepting the language $L_\omega(M_1) \cap L_\omega(M_2)$. The proof is based on a product construction for Büchi-automata, see e.g. [18]. For ω-words $w_1, \ldots, w_n \in \Gamma^\omega$, the *convolution* $w_1 \otimes w_2 \otimes \cdots \otimes w_n \in (\Gamma^n)^\omega$ is

$$w_1 \otimes \cdots \otimes w_n = (w_1(1), \ldots, w_n(1))\,(w_1(2), \ldots, w_n(2))\,(w_1(3), \ldots, w_n(3)) \cdots$$

An n-ary relation $R \subseteq (\Gamma^\omega)^n$ is called *ω-automatic* if the language $\{w_1 \otimes \cdots \otimes w_n \mid (w_1, \ldots, w_n) \in R\}$ is a regular ω-language, i.e., accepted by some n-dimensional Büchi-automaton.

ω-Automatic structures. A *signature* is a finite set τ of relational symbols, where each relational symbol $R \in \tau$ has an associated arity n_R. A (relational) *structure over the signature* τ, briefly a *τ-structure*, is a tuple $\mathcal{A} = (A, (R^{\mathcal{A}})_{R \in \tau})$, where A is a set (the universe of $\mathcal{A}$) and $R^{\mathcal{A}}$ is a relation of arity n_R over the set A, which interprets the relational symbol R. We will assume that every signature contains the equality symbol $=$ and that $=^{\mathcal{A}}$ is the identity relation on the set A. Usually, we denote the relation $R^{\mathcal{A}}$ also with R. We will also write $a \in \mathcal{A}$ for $a \in A$. For a subset $B \subseteq A$ we denote with $\mathcal{A}{\restriction}B$ the restriction $(B, (R^{\mathcal{A}} \cap B^{n_R})_{R \in \tau})$.

Let $\mathcal{A}$ be an arbitrary τ-structure with universe A. An *injectively ω-automatic presentation* for $\mathcal{A}$ is a tuple (Γ, L, h) such that

- Γ is a finite alphabet,
- $L \subseteq \Gamma^\omega$ is a regular ω-language,
- $h : L \to A$ is a bijection, and
- the relation $\{(u_1, \ldots, u_{n_R}) \in L^{n_R} \mid (h(u_1), \ldots, h(u_{n_R})) \in R\}$ is ω-automatic for every $R \in \tau$.

The structure $\mathcal{A}$ is injectively *ω-automatic* if there is an injectively ω-automatic presentation for $\mathcal{A}$. A typical example of an injectively ω-automatic structure is $(\mathbb{R}, +)$.

Remark 2.1. The original definition of an ω-automatic presentation requires h to be only surjective and the relation $\{(u, v) \in L^2 \mid h(u) = h(v)\}$ to be ω-automatic [4]. In [4, Proposition 5.2] it is claimed that every ω-automatic structure (according to this original definition) has an injectively ω-automatic presentation. The following example shows that the proof of [4, Proposition 5.2] does not work: Let two sets A and B of natural numbers be equivalent ($A \approx B$) if and only if the symmetric difference $A \triangle B$ is finite. Then the quotient $\mathcal{B}$ of the power-set of $\mathbb{N}$ wrt. $\approx$ is a Boolean algebra. It has an ω-automatic presentation in the more general sense of [4] with underlying set $L = \{0, 1\}^\omega$ and $h(w) = [\{i \in \mathbb{N} \mid w(i) = 1\}]_\approx$. But there is no ω-regular subset $K \subseteq L$ such that, for any $u \in L$, there is precisely one $v \in K$ with $h(u) = h(v)$, as was claimed in [4]. It is therefore open, whether every ω-automatic structure (in the original sense) has an injectively ω-automatic presentation. Since this paper deals with injectively ω-automatic structures exclusively, we will always assume an injectively ω-automatic presentation (Γ, L, h), where L is the universe of the structure and h is the identity function. Furthermore, we use the more concise notation "ω-automatic presentation" (resp. "ω-automatic structure") instead of "injectively ω-automatic presentation" (resp. "injectively ω-automatic structure").

Automatic structures are defined in the same way as ω-automatic structures, except that finite automata over finite words instead of Büchi-automata are used (the convolution of finite words requires an additional letter $\perp$ that is appended to the arguments in order to make them the same length). By [3, Theorem 5.32], a countable structure is automatic if and only if it is ω-automatic.

Logic. In addition to the usual first-order quantifier $\exists$, this section is concerned with quantifiers $\exists^\infty$, $\exists^\varkappa$ for a cardinal $\varkappa$, and $\exists^{(t,k)}$ for $0 \leq t < k > 1$ two natural numbers. The semantics of these quantifiers are defined as follows:

- $\mathcal{A} \models \exists^\infty x\, \psi$ if and only if there are infinitely many $a \in \mathcal{A}$ with $\mathcal{A} \models \psi(a)$.
- $\mathcal{A} \models \exists^\varkappa x\, \psi$ if and only if the set $\{a \in \mathcal{A} \mid \mathcal{A} \models \psi(a)\}$ has cardinality $\varkappa$.
- $\mathcal{A} \models \exists^{(t,k)} x\, \psi$ if and only if the set $\{a \in \mathcal{A} \mid \mathcal{A} \models \psi(a)\}$ is finite and $t = |\{a \in \mathcal{A} \mid \mathcal{A} \models \psi(a)\}| \bmod k$.

We will denote by FO the set of first-order formulas. For a class of cardinals C, $\mathrm{FO}(\exists^\infty, (\exists^\varkappa)_{\varkappa \in C}, (\exists^{(t,k)})_{0 \leq t < k > 1})$ is the set of formulas using $\exists$ and the quantifiers listed. For any set $\mathcal{L}$ of formulas, the *$\mathcal{L}$-theory* of a structure $\mathcal{A}$ is the set of sentences (i.e., formulas without free variables) from $\mathcal{L}$ that hold in $\mathcal{A}$. The following result can be shown by induction on the structure of the formula φ.

Proposition 2.2 (cf. [4, 11, 13]). *Let (Γ, L, h) be an automatic presentation for the structure $\mathcal{A}$ and let $\varphi(x_1, \ldots, x_n)$ be a formula of* $\mathrm{FO}(\exists^\infty, (\exists^{(t,k)})_{0\le t<k\ge 2})$ *over the signature of $\mathcal{A}$. Then the relation*

$$\{(u_1, \ldots, u_n) \in L^n \mid \mathcal{A} \models \varphi(h(u_1), \ldots, h(u_n))\}$$

is effectively automatic. It is effectively ω-automatic if (Γ, L, h) is an ω-automatic presentation for the structure $\mathcal{A}$ and φ belongs to $\mathrm{FO}(\exists^\infty)$.

This theorem implies the following result, which is one of the main motivations for investigating (ω-)automatic structures.

Theorem 2.3 ([4, 13]). *If $\mathcal{A}$ is an ω-automatic structure, then its* $\mathrm{FO}(\exists^\infty)$*-theory is decidable. If $\mathcal{A}$ is automatic, then even its* $\mathrm{FO}(\exists^\infty, (\exists^{(t,k)})_{0\le t<k\ge 2})$*-theory is decidable.*

Note that any automatic structure $\mathcal{A}$ is at most countably infinite. Hence the quantifiers $\exists^\infty$ and $\exists^{\aleph_0}$ are equivalent in this setting. Furthermore, no formula $\exists^\varkappa x\, \psi$ with $\varkappa > \aleph_0$ holds in $\mathcal{A}$. Hence, for any countable set of cardinals C, the $\mathrm{FO}(\exists^\infty, (\exists^\varkappa)_{\varkappa\in C}, (\exists^{(t,k)})_{0\le t<k>1})$-theory of an automatic structure is decidable.[2] In the rest of Section 2 we extend this result to ω-automatic structures.

To the knowledge of the authors, the modulo quantifiers $\exists^{(t,k)}$ have not yet been considered for ω-automatic structures. Since an ω-automatic structure can have up to $2^{\aleph_0}$ many elements, it makes sense to consider quantifiers of the form $\exists^\varkappa$ with $\aleph_0 \le \varkappa \le 2^{\aleph_0}$.

2.2 Cardinality and Modulo Quantifiers for ω-Automatic Structures

It is the aim of this section to extend the realm of Proposition 2.2 and therefore of Theorem 2.3 to ω-automatic structures. To this aim, we fix an ω-automatic structure $\mathcal{A}$ with presentation (Γ, L, id).

Two infinite words v and w are *ultimately equal*, briefly $v \sim w$, if there exists $i \in \mathbb{N}$ such that $v(j) = w(j)$ for $j \ge i$. Since the relation $\sim$ is ω-automatic, we can assume it to be among the relations of the ω-automatic structure $\mathcal{A}$. The following lemma is our main combinatorial tool for analyzing ω-automatic structures.

Lemma 2.4. *Let M be a Büchi-automaton with n states over $\Sigma \times \Gamma$, $u \in \Sigma^\omega$, and $V = \{v \in \Gamma^\omega \mid u \otimes v \in L_\omega(M)\}$. Then $|V| = 2^{\aleph_0}$ if and only if $|V/{\sim}| > n$. Moreover, $|V| \in \mathbb{N} \cup \{\aleph_0, 2^{\aleph_0}\}$ and the exact number can be computed in polynomial space.*

This lemma allows to handle the quantifiers $\exists^{\aleph_0}$ and $\exists^{2^{\aleph_0}}$:

Proposition 2.5. *Let the relation $R \subseteq (\Gamma^\omega)^{n+1}$ be ω-automatic. Then the relation $R_\varkappa = \{(u_1, \ldots, u_n) \mid \mathcal{A} \models \exists^\varkappa x_{n+1} : R(u_1, \ldots, u_n, x_{n+1})\}$ is effectively ω-automatic for $\varkappa \in \{\aleph_0, 2^{\aleph_0}\}$.*

[2] C has to be countable for otherwise the set of formulas would become uncountable rendering the decidability question nonsense.

Proof. Let the convolution of R be accepted by an $(n+1)$-dimensional Büchi-automaton with m states. Then the formula $\exists^{2^{\aleph_0}} x_{n+1} : R(u_1, \dots, u_n, x_{n+1})$ is, by Lemma 2.4, equivalent with

$$\mathcal{A} \models \exists x_0 \cdots \exists x_m \left\{ \bigwedge_{0 \le i < j \le m} x_i \not\sim x_j \wedge \bigwedge_{0 \le i \le m} R(u_1, \dots, u_n, x_i) \right\}.$$

Lemma 2.4 also ensures that the quantifier $\exists^{\aleph_0}$ is equivalent with saying "there are infinitely, but not $2^{\aleph_0}$ many". Hence, by Proposition 2.2, $R_\varkappa$ is ω-regular. □

We now want to prove the corresponding result for modulo quantifiers. As above, let $R \subseteq (\Gamma^\omega)^{n+1}$ be ω-automatic and let $0 \le t < k \ge 2$. Because of Proposition 2.5, we can assume that for all $u_1, \dots, u_n \in \Gamma^\omega$, there are only finitely many $v \in \Gamma^\omega$ with $(u_1, \dots, u_n, v) \in R$.

For the following, it is convenient to write $\Sigma = \Gamma^n$ and consider R as an ω-automatic subset of $\Sigma^\omega \times \Gamma^\omega$. Since the convolution of R is ω-regular, it can be accepted by some *deterministic* Muller-automaton $M = (Q, \Sigma \times \Gamma, \delta, \iota, \mathcal{F})$ (see e.g. [18] for details concerning Muller automata). Now consider the alphabet $\Delta = \Sigma \times \Gamma \times \{0, \dots, k-1\}^Q \times \{0,1\}^Q$. Then one can construct a Büchi-automaton M' over Δ that accepts an ω-word $(a_i, b_i, f_i, g_i)_{i \ge 0} \in \Delta^\omega$ if and only if we have for all $i \ge 0$ and all $p \in Q$:

(1) $f_i(p) = |\{w \in \Gamma^* \mid |w| = i, \delta(\iota, a_0 a_1 \cdots a_{i-1} \otimes w) = p\}| \bmod k$ (i.e., $f(p)$ is the number of possible partners modulo k that allow $a_0 \cdots a_{i-1}$ to move from the initial state of M into p)
(2) $g_i(p) = 1$ if and only if the ω-word $a_i a_{i+1} \cdots \otimes b_i b_{i+1} \cdots$ has an accepting run in M from the state p.

To ensure condition (1), one actually constructs an automaton that counts the number of runs from ι to p whose label is of the form $a_0 a_1 \cdots a_{i-1} \otimes w$ for some word w. Since the automaton M is deterministic, this number equals the number of partners as desired.

Note that for any $u \in \Sigma^\omega$ and $v \in \Gamma^\omega$, there is precisely one ω-word $x \in L(M')$ whose projection $\pi(x)$ onto $(\Sigma \times \Gamma)^\omega$ equals $u \otimes v$.

Lemma 2.6. *Let $u \in \Sigma^\omega$ and $v \in \Gamma^\omega$, and let $x = (a_i, b_i, f_i, g_i)_{i \in \omega} \in \Delta^\omega$ be the unique ω-word with $\pi(x) = u \otimes v$. There is $i \in \mathbb{N}$ such that for all $j \ge i$, we have*

$$\sum \{f_j(p) \mid p \in Q, g_j(p) = 1\} \equiv |\{w \in \Gamma^\omega \mid w \sim v, (u, w) \in R\}| \bmod k. \quad (1)$$

Note that by our assumption on R, the set $\{w \in \Gamma^\omega \mid w \sim v, (u, w) \in R\}$ is always finite, hence the expression makes sense. The lemma thus says that the sum on the left is eventually fix and gives the number of possible parters w of u that are ultimately equal to v. From the Büchi-automaton M', we can build a new Büchi-automaton M'_s (for $0 \le s < k$) over Δ that checks whether the sum on the left in (1) is eventually fix and equal s. Let M_s be the projection

of the automaton M'_s to the alphabet $\Sigma \times \Gamma$. Then M_s accepts $u \otimes v$ if and only if, modulo k, there are s many ω-words w ultimately equal to v such that $(u, w) \in R$.

Since R is ω-automatic, there is a Büchi-automaton with, say, m states accepting the convolution of R. Let $u = (u_1, \dots, u_n) \in \Sigma^\omega$. Since, by our assumption on R, the set $\{v \in \Gamma^\omega \mid (u, v) \in R\}$ is finite, there are $r \leq m$ many ω-words $v_1, \dots, v_r$ in this set that are mutually not ultimately equal (Lemma 2.4). Thus, we have $\exists^{(t,k)} x_{n+1} : R(u_1, \dots, u_n, x_{n+1})$ if and only if there exist $r \leq m$, mutually not ultimately equal words $v_1, \dots, v_r \in \Sigma^\omega$, and integers $0 \leq t_i < k$ for $1 \leq i \leq r$ such that

1. $R(u_1, \dots, u_n, v_i)$ for $1 \leq i \leq r$,
2. for any $v \in \Sigma^\omega$ with $R(u_1, \dots, u_n, v)$, there exists i with $v \sim v_i$,
3. $t = \sum_{i=1}^{r} t_i \bmod k$ and $u_1 \otimes \cdots \otimes u_n \otimes v_i \in L_\omega(M_{t_i})$ for $1 \leq i \leq r$.

Since m is a constant depending on R, only, these conditions can be expressed in first-order logic. Hence Proposition 2.2 implies that $\{(u_1, \dots, u_n) \mid \exists^{(t,k)} x_{n+1} : R(u_1, \dots, u_n, x_{n+1})\}$ is ω-automatic. Thus, we showed:

Proposition 2.7. *Let the relation $R \subseteq (\Gamma^\omega)^{n+1}$ be ω-automatic and let $0 \leq t < k \geq 2$. Then the relation $\{(u_1, \dots, u_n) \mid \mathcal{A} \models \exists^{(t,k)} x_{n+1} : R(u_1, \dots, u_n, x_{n+1})\}$ is effectively ω-automatic.*

Together with Propositions 2.2 and 2.5, we obtain:

Theorem 2.8. *Let $\mathcal{A}$ be an ω-automatic structure and let C be an at most countably infinite set of cardinals. Then the* $\mathrm{FO}(\exists^\infty, (\exists^\varkappa)_{\varkappa \in C}, (\exists^{(t,k)})_{0 \leq t < k > 1})$-*theory of $\mathcal{A}$ is decidable.*

Proof. Lemma 2.4 implies that a formula of the form $\exists^\varkappa x\, \psi$ with $\aleph_0 < \varkappa < 2^{\aleph_0}$ can never be true in $\mathcal{A}$. Hence, the theory in question can be reduced to the $\mathrm{FO}(\exists^\infty, \exists^{\aleph_0}, \exists^{2^{\aleph_0}}, (\exists^{(t,k)})_{0 \leq t < k > 1})$-theory of $\mathcal{A}$. Since emptiness of Büchi-automata is decidable, the result follows from Propositions 2.2, 2.5, and 2.7. □

3 ω-Automatic Structures of Bounded Degree and Complexity of Theories

As first observed in [4], there are automatic structures with a non-elementary first-order theory. Our aim in this section is to single out a class of ω-automatic structures such that the $\mathrm{FO}(\exists^\infty, \exists^{\aleph_0}, \exists^{2^{\aleph_0}}, (\exists^{(t,k)})_{0 \leq t < k > 1})$-theory is elementarily decidable. In doing so, we will find that even more general quantifiers give rise to elementarily decidable theories provided we constrain ourselves to structures of bounded degree.

3.1 Definitions and Known Results

Structures of bounded degree. Let $\mathcal{A}$ be a τ-structure with universe A. The *Gaifman-graph* $G_\mathcal{A}$ of the structure $\mathcal{A}$ is the following undirected graph:

$$G_\mathcal{A} = (A, \{(a, b) \in A \times A \mid \exists R \in \tau\, \exists (c_1, \dots, c_{n_R}) \in R\, \exists j, k : c_j = a \neq b = c_k\}).$$

Thus, the set of nodes is the universe of $\mathcal{A}$ and there is an edge between two elements, if and only if they are contained in some tuple belonging to one of the relations of $\mathcal{A}$. The structure $\mathcal{A}$ is *locally finite*, if every node of the Gaifman-graph $G_{\mathcal{A}}$ has only finitely many neighbors. It has *bounded degree*, if its Gaifman-graph $G_{\mathcal{A}}$ has bounded degree, i.e., there exists a constant d such that every $a \in A$ is adjacent to at most d other nodes in $G_{\mathcal{A}}$.

In contrast to the general case, if the degree of the automatic structure $\mathcal{A}$ is bounded, an elementary upper bound for the first-order theory of $\mathcal{A}$ is due to the second author (we define $\exp(1,n) = 2^n$ and $\exp(k+1,n) = 2^{\exp(k,n)}$):

Theorem 3.1 ([16]). *If $\mathcal{A}$ is an automatic structure of bounded degree, then the* FO*-theory of $\mathcal{A}$ can be decided in* $\mathrm{SPACE}(\exp(3, O(n)))$ *and there is such a structure for which* $\mathrm{SPACE}(\exp(2, O(n)))$ *is a lower bound.*

This result was not known to apply to more general quantifiers nor to ω-automatic structures. An important tool in the proof of Theorem 3.1 as well as in our extension, is the concept of a sphere that we introduce next.

With $d_{\mathcal{A}}(a,b)$, where $a,b \in A$, we denote the distance between a and b in $G_{\mathcal{A}}$, i.e., it is the length of a shortest path connecting a and b in $G_{\mathcal{A}}$. For $a \in A$ and $r \geq 0$ we denote with $S_{\mathcal{A}}(r,a) = \{b \in A \mid d_{\mathcal{A}}(a,b) \leq r\}$ the *r-sphere around a*. If $\bar{a} = (a_1,\ldots,a_n) \in A^n$ is a tuple, then $S_{\mathcal{A}}(r,\bar{a}) = \bigcup_{i=1}^{n} S_{\mathcal{A}}(r,a_i)$. The *neighborhood* $N_{\mathcal{A}}(r,\bar{a}) = \mathcal{A}\restriction S_{\mathcal{A}}(r,\bar{a})$ of radius r around $\bar{a}$ is the substructure of $\mathcal{A}$ induced by $S_{\mathcal{A}}(r,\bar{a})$.

Generalized quantifiers and locality. Let us fix a relational signature τ. In this section, we will consider the logic $\mathcal{L}(\mathcal{Q}_u)$. Formulas of the logic $\mathcal{L}(\mathcal{Q}_u)$ are built from atomic formulas of the form $R(x_1,\ldots,x_{n_R})$, where $R \in \tau$ is a relational symbol and $x_1,\ldots,x_{n_R}$ are first-order variables ranging over the universe of the underlying structure, using boolean connectives and quantifications of the form $\mathcal{Q}_{\mathcal{C}} y : (\psi_1(\bar{x},y),\ldots,\psi_n(\bar{x},y))$. Here, $\psi_i(\bar{x},y)$ is already a formula of $\mathcal{L}(\mathcal{Q}_u)$, $\bar{x}$ is a sequence of variables, and $\mathcal{C}$ is an n-ary relation over cardinals, i.e., $\mathcal{C} = \{(\varkappa_{i,1},\ldots,\varkappa_{i,n}) \mid i \in J, \varkappa_{i,j} \text{ is a cardinal}\}$ for some index set J. To define the semantics of the $\mathcal{Q}_{\mathcal{C}}$-quantifier, let $\mathcal{A}$ be a τ-structure with universe A and let $\bar{u}$ be a tuple of values from A of the same length as $\bar{x}$. Then $\mathcal{A} \models \mathcal{Q}_{\mathcal{C}} y : (\psi_1(\bar{u},y),\ldots,\psi_n(\bar{u},y))$ if and only if $(\varkappa_1,\ldots,\varkappa_n) \in \mathcal{C}$, where $\varkappa_i$ is the cardinality of the set $\{a \in A \mid \mathcal{A} \models \psi_i(\bar{u},a)\}$. In the above situation, we call the quantifier $\mathcal{Q}_{\mathcal{C}}$ also an *n-dimensional counting quantifier*. The *quantifier rank* $\mathrm{qfr}(\varphi)$ of a formula φ is the maximal number of nested quantifiers of φ. The logic $\mathcal{L}(\mathcal{Q}_u)$ is a finitary fragment of the logic $\mathcal{L}_{\infty,\omega}(\mathcal{Q}_u)^{\omega}$ from [10], which allows infinite conjunctions and disjunctions but restricts to finite quantifier rank.

Let us consider some examples for generalized quantifiers. The ordinary existential quantifier $\exists y : \varphi(\bar{x},y)$ is equivalent to $\mathcal{Q}_{\mathcal{C}}\, y : \varphi(\bar{x},y)$, where $\mathcal{C}$ is the class of all non-zero cardinals. Similarly, we can obtain the counting quantifier $C_K\, y : \varphi(\bar{x},y)$ for K some class of cardinals ("the number of y satisfying $\varphi(\bar{x},y)$ belongs to K"). Well-known special cases of the latter quantifier are the quantifiers $\exists^{\infty}$, $\exists^{\varkappa}$, and $\exists^{(t,q)}$ from the Section 2. All these counting quantifiers are one-dimensional. A well-known two-dimensional counting quantifier is the

Härtig quantifier $I\,y : (\psi_1(\bar{x}, y), \psi_2(\bar{x}, y))$ ("the number of y satisfying $\psi_1(\bar{x}, y)$ equals the number of y satisfying $\psi_2(\bar{x}, y)$"). For this we have to choose for $\mathcal{C}$ the identity relation on cardinals.

For a class $\mathbb{C}$, where every $\mathcal{C} \in \mathbb{C}$ is a relation on cardinals, FO($\mathbb{C}$) denotes those formulas of $\mathcal{L}(\mathcal{Q}_u)$ that only use quantifiers of the form $\mathcal{Q}_\mathcal{C}$ with $\mathcal{C} \in \mathbb{C}$ along with the existential quantifier $\exists$. For a singleton class $\mathbb{C} = \{\mathcal{C}\}$ we also write FO($\mathcal{C}$) instead of FO($\mathbb{C}$).

We will make use of the following locality principle for the logic $\mathcal{L}(\mathcal{Q}_u)$:

Theorem 3.2 ([10]). *Let $\mathcal{A}$ be a locally finite structure, let $\varphi(x_1, \ldots, x_k)$ be an $\mathcal{L}(\mathcal{Q}_u)$-formula of quantifier rank at most d, and let $\bar{a}, \bar{b} \in \mathcal{A}^k$ be k-tuples with $(N_\mathcal{A}(2^d, \bar{a}), \bar{a}) \cong (N_\mathcal{A}(2^d, \bar{b}), \bar{b})$.*[3] *Then $\mathcal{A} \models \varphi(\bar{a})$ if and only if $\mathcal{A} \models \varphi(\bar{b})$.*

Proof. Keisler and Lotfallah [10] proved this statement for locally finite *countable* structures. As an intermediate step, they considered an infinitary logic with counting quantifiers C_A with $A = \{0, 1, 2, \ldots, n\}$ for some $n \in \mathbb{N}$. Considering, instead, counting quantifiers C_A with $A = \{\lambda \mid \lambda \leq \varkappa\}$ for $\varkappa$ a cardinal, one obtains the above general theorem (which does not restrict to countable structures) without any further modifications of [10]. □

3.2 Complexity of the $\mathcal{L}(Q_u)$-Theory

In Section 3.4 we will show that there exists a locally finite automatic structure $\mathcal{A}$ and a recursive set $K \subseteq \mathbb{N}$ such that the FO(C_K)-theory of $\mathcal{A}$ is undecidable. To obtain a decidability result, we therefore consider an ω-automatic structure $\mathcal{A}$ of bounded degree. We will consider the FO($\mathbb{C}$)-theory of $\mathcal{A}$, where every $\mathcal{C} \in \mathbb{C}$ is a relation over cardinals. Furthermore, we make the following assumptions:

(1) (Γ, L, id) is an ω-automatic presentation for $\mathcal{A}$, i.e., in particular L is the universe of $\mathcal{A}$.
(2) $\delta \in \mathbb{N}$ is a bound for the degrees of the nodes in the Gaifman graph $G_\mathcal{A}$.
(3) For every $0 \leq n \leq \delta$ the signature τ contains a unary predicate $\deg_n$ with $\mathcal{A} \models \deg_n(u)$ if and only if the degree of u in the Gaifman-graph $G_\mathcal{A}$ is exactly n.
(4) $\mathbb{C}$ is a countable set of relations on $\mathbb{N} \cup \{\aleph_0, 2^{\aleph_0}\}$.

Clearly, neither (1) nor (2) imposes restrictions on (the isomorphism type of) $\mathcal{A}$. Since the set of nodes w of degree n is first-order definable, it is ω-regular. Hence we can assume it to be among the relations of $\mathcal{A}$. Thus, (3) is no essential restriction. Finally, consider (4). If $\mathbb{C}$ allows more than countably many relations, then it does not make sense to ask for the decidability of the FO($\mathbb{C}$)-theory of $\mathcal{A}$ since it is uncountable. Furthermore, one can show that even without restricting to relations over $\mathbb{N} \cup \{\aleph_0, 2^{\aleph_0}\}$, the size of any definable set belongs to $\mathbb{N} \cup \{\aleph_0, 2^{\aleph_0}\}$. Hence we can safely assume (4).

We will prove that under the above four restrictions, the FO($\mathbb{C}$)-theory of $\mathcal{A}$ can be reduced in triply exponential space to the relations in $\mathbb{C}$. For this, we need

[3] This means that there exists an isomorphism $f : N_\mathcal{A}(2^d, \bar{a}) \to N_\mathcal{A}(2^d, \bar{b})$ mapping for every $1 \leq i \leq k$ the i-th entry of $\bar{a}$ to the i-th entry of $\bar{b}$.

the following concept: A pair $(\mathcal{B}, \bar{b})$ is a *potential* (D,k)*-sphere* (for $D, k \in \mathbb{N}$) if the following holds:

- $\mathcal{B}$ is a finite τ-structure whose Gaifman-graph has degree at most δ,
- $\bar{b}$ is a k-tuple of elements from $\mathcal{B}$,
- $N_{\mathcal{B}}(2^D, \bar{b}) = \mathcal{B}$, i.e., every element of $\mathcal{B}$ has distance at most 2^D from some entry of the tuple $\bar{b}$,
- for any $y \in S_{\mathcal{B}}(2^D - 1, \bar{b})$, we have $\mathcal{B} \models \deg_n(y)$ if and only if n is the degree of y in the Gaifman-graph of $\mathcal{B}$, and
- for any $y \in \mathcal{B} \setminus S_{\mathcal{B}}(2^D - 1, \bar{b})$ there is a unique $0 \leq n \leq \delta$ such that $\mathcal{B} \models \deg_n(y)$ and the degree of y in the Gaifman-graph of $\mathcal{B}$ is at most n.

Thus, a potential (D,k)-sphere is a candidate for a 2^D-sphere around some k-tuple in the structure $\mathcal{A}$.

Let $\{b_1, b_2, \ldots, b_n\}$ be the universe of $\mathcal{B}$ with $\bar{b} = (b_1, \ldots, b_k)$ $(k \leq n)$. Since $\bar{b}$ is not necessarily repetition-free, we may have $b_i = b_j$ for $i \neq j$ in case $i, j \leq k$, but we may assume that $b_{k+1}, \ldots, b_n$ are pairwise different and different from $b_1, \ldots, b_k$. We define $\varphi_{(\mathcal{B},\bar{b})}(x_1, \ldots, x_k) = \exists x_{k+1} \cdots \exists x_n : \psi(x_1, \ldots, x_n)$, where $\psi(x_1, \ldots, x_n)$ is the conjunction of the following formulas:

- $x_i = x_j$ if $b_i = b_j$ and $x_i \neq x_j$ if $b_i \neq b_j$
- $R(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ if $(b_{i_1}, b_{i_2}, \ldots, b_{i_m}) \in R$ for $R \in \tau$ with $m = n_R$ and $i_1, \ldots, i_m \in \{1, \ldots, n\}$
- $\neg R(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ if $(b_{i_1}, b_{i_2}, \ldots, b_{i_m}) \notin R$ for $R \in \tau$ with $m = n_R$ and $i_1, \ldots, i_m \in \{1, \ldots, n\}$.

Lemma 3.3. *There exists a constant* $c \in \mathbb{N}$ *such that for any potential* (D,k)*-sphere* $(\mathcal{B}, \bar{b})$*, the existential FO-formula* $\varphi_{(\mathcal{B},\bar{b})}$ *has size at most* $\exp(2, c(D+k))$*. For any k-tuple* $\bar{u} \in L^k$*, we have:* $\mathcal{A} \models \varphi_{(\mathcal{B},\bar{b})}(\bar{u}) \Leftrightarrow (N_{\mathcal{A}}(2^D, \bar{u}), \bar{u}) \cong (\mathcal{B}, \bar{b})$.

Lemma 3.4. *There are functions* $\# : \mathbb{N}^2 \to \mathbb{N}$ *and* $\Phi : \mathbb{N}^3 \to \mathrm{FO}$ *such that*

1. $\#(D,k)$ *is computable in space* $\exp(2, O(D+k))$ *and* $\Phi(D,k,i)$ *in space* $\exp(2, O(D+k)) + \log(i)$
2. *for any* $D, k \in \mathbb{N}$*,* $\#(D,k)$ *is the number of potential* (D,k)*-spheres,*
3. *for any* $D, k, i \in \mathbb{N}$*, there exists a potential* (D,k)*-sphere* $\mathcal{B}(D,k,i)$ *with* $\varphi_{\mathcal{B}(D,k,i)} = \Phi(D,k,i)$*, and*
4. *for any* $D, k \in \mathbb{N}$ *and any potential* (D,k)*-sphere* $(\mathcal{B}, \bar{b})$*, there exists* $1 \leq i \leq \#(D,k)$ *with* $\varphi_{(\mathcal{B},\bar{b})} = \Phi(D,k,i)$.

Note that $\mathcal{B}(D,k,1), \ldots, \mathcal{B}(D,k,\#(D,k))$ enumerates the isomorphism types of potential (D,k)-spheres for any $D, k \in \mathbb{N}$.

In the following we identify a tuple $\bar{u} = (u_1, \ldots, u_k)$ with its convolution $u_1 \otimes u_2 \otimes \cdots \otimes u_k$. We write $k = |\bar{u}|$ for the length of the tuple $\bar{u}$.

Lemma 3.5. *The following can be computed in space* $\exp(3, O(D+k)) + \log(i)$*:*
INPUT: $D, k, i \in \mathbb{N}$
QUTPUT: a k-dimensional Büchi-automaton M of size $\exp(3, O(D+k))$ *with*
$$L_\omega(M) = \{\bar{u} \mid (N_{\mathcal{A}}(2^D, \bar{u}), \bar{u}) \cong \mathcal{B}(D,k,i)\}.$$

Let us fix a function $s(D+k) \in \exp(3, O(D+k))$ bounding the space in Lemma 3.5. For a word $u \in \Sigma^\omega$, its *norm* $\lambda(u)$ is $\lambda(u) = \inf\{|vw| \mid u = vw^\omega\}$, with $\lambda(u) = \infty$ if u is not ultimately periodic, i.e., not of the form vw^ω for some $v, w \in \Sigma^*$. Let UP denote the class of all ultimately periodic ω-words over some alphabet. In the algorithms below, we will often handle ω-words $u \in \mathrm{UP}$ that can be given as a pair (v, w) with $u = vw^\omega$ and $|vw| = \lambda(w)$. Note that if M is a Büchi-automaton with n states and $L_\omega(M) \neq \emptyset$, then we find an ω-word $u \in L_\omega(M)$ such that $\lambda(u) \leq 2n$. Note that for $\bar{u} = (u_1, \ldots, u_k)$ we have $\lambda(\bar{u}) = \lambda(u_1 \otimes u_2 \cdots \otimes u_k) \leq \prod_{1 \leq i \leq k} \lambda(u_i)$. Since we can build a $(k+1)$-dimensional Büchi-automaton with $\lambda(\bar{u})$ many states that accepts the language $\bar{u} \otimes \Sigma^\omega$, the product construction for Büchi-automata and Lemma 3.5 gives:

Lemma 3.6. *The following can be computed in space* $3 \cdot s(D+k+1) \cdot \lambda(\bar{u}) + \log(i)$ *if* $k = |\bar{u}| > 0$ *and in space* $s(D+1) + \log(i)$ *if* $k = |\bar{u}| = 0$*:*

INPUT: $D, k, i \in \mathbb{N}$ *and* $\bar{u} \in L^k \cap \mathrm{UP}$
OUTPUT: a $(k+1)$*-dimensional Büchi-automaton* M *with* $L_\omega(M) = \{\bar{u}w \in L^{k+1} \mid (N_{\mathcal{A}}(2^D, \bar{u}w), \bar{u}w) \cong \mathcal{B}(D, k+1, i)\}$.
Moreover, if $L_\omega(M) \neq \emptyset$*, then we can compute within the same space bound a word* $w \in L \cap \mathrm{UP}$ *with* $\bar{u}w \in L_\omega(M)$ *and*

$$\lambda(w) \leq \begin{cases} 6 \cdot s(D+k+1) \cdot \lambda(\bar{u}) & \text{if } k > 0 \\ 2 \cdot s(D+1) & \text{if } k = 0. \end{cases} \qquad (*)$$

Now consider the following two algorithms **size** and **check**. The algorithm **size** shall return the number of words $v \in \Sigma^\omega$ with $\mathcal{A} \models \varphi(\bar{u}v)$. The algorithm **check** shall check whether $\mathcal{A} \models \varphi(\bar{u})$.

```
1   check(φ(x̄), ū) : {0,1}
2         (φ(x̄) formula with |ū| = |x̄| many free variables,
3         ū tuple of ultimately periodic words from L)
4     case φ = R(x̄)
5       if ū ∈ R then return(1) else return(0) endif
6     case φ = φ1 ∧ φ2
7       return(check(φ1, ū) ∧ check(φ2, ū))
8     case φ = ¬φ1
9       return(¬check(φ1, ū))
10    case φ = Q_C y : (ψ1(x̄, y), ..., ψn(x̄, y))
11      for  i = 1  to  n do
12        ϰi := size(ψi, ū)
13      endfor
14      if (ϰ1, ..., ϰn) ∈ C  then return(1) else return(0) endif
```

```
1   size(φ, ū) : ℕ ∪ {ℵ0, 2^ℵ0}
2         (φ formula with |ū| + 1 many free variables,
3         ū tuple of ultimately periodic words from L)
4     D := qfr(φ); ϰ := 0;
```

5 **for** $i := 1$ **to** $\#(D, |\bar{u}|+1)$ **do**
6 **calculate** an $|\bar{u}|+1$-dimensional Büchi-automaton M with
$L_\omega(M) = \{\bar{u}w \in L^{|\bar{u}|+1} \mid (N_{\mathcal{A}}(2^D, \bar{u}w), \bar{u}w) \cong \mathcal{B}(D, |\bar{u}|+1, i)\}$
7 **if** $L_\omega(M) \neq \emptyset$ **then**
8 **choose** $w \in \Sigma^\omega$ with $\bar{u}w \in L_\omega(M)$ and $\lambda(w) \leq 2 \cdot s(D+1)$
9 if $|\bar{u}| = 0$ and $\lambda(w) \leq 6 \cdot s(D+|\bar{u}|+1) \cdot \lambda(\bar{u})$ otherwise
10 **if check**$(\varphi, \bar{u}w)$ **then**
11 $\varkappa := \varkappa + |L_\omega(M)|$
12 **endif**
13 **endif**
14 **endfor**
15 **return**$(\varkappa)$

Let us first verify the correctness of these algorithms. If **size** behaves as intended, the correctness of **check** is rather obvious. We now discuss **size**. By Lemma 3.4, line 5 iterates over all potential $(D, |\bar{u}|+1)$-spheres. Since $D = \mathrm{qfr}(\varphi)$, there *exists* a tuple $\bar{u}w \in L_\omega(M)$ with $\mathcal{A} \models \varphi(\bar{u}w)$ if and only if $\mathcal{A} \models \varphi(\bar{u}v)$ for *all* $\bar{u}v \in L_\omega(M)$ by Theorem 3.2, where M is the Büchi-automaton calculated in line 6. Therefore, we select in line 8,9 a "short" tuple $\bar{u}w \in L_\omega(M)$ and check in line 10 whether $\mathcal{A} \models \varphi(\bar{u}w)$ using algorithm **check**. If this is true, then we add to the current $\varkappa$ the size of the language $L_\omega(M)$, which can be calculated by Lemma 2.4 in polynomial space wrt. the size of M.

Next we discuss the space complexity of a call **check**(ψ, ε) (where ε is the empty tuple) for a sentence ψ of quantifier rank D_0. Note that when we call **size** with parameters φ and $\bar{u}$, then $\mathrm{qfr}(\varphi) + |\bar{u}| + 1 \leq D_0$. Thus, the Büchi-automaton M in line 6 can be calculated in space $3 \cdot s(D+|\bar{u}|+1) \cdot \lambda(\bar{u}) \leq 3 \cdot s(D_0) \cdot \lambda(\bar{u})$ by Lemma 3.6 (since $i \leq \#(D, |\bar{u}|+1) \in \exp(3, O(D_0))$, we can forget the summand $\log(i)$) and also the bound $2 \cdot s(D+1) \leq 2 \cdot s(D_0)$ (resp. $6 \cdot s(D+|\bar{u}|+1) \cdot \lambda(\bar{u}) \leq 6 \cdot s(D_0) \cdot \lambda(\bar{u})$) in line 8,9 for the ω-word w follows from Lemma 3.6. Assume that $(u_1, u_2, \ldots, u_{D_0})$ is the tuple of ultimately periodic ω-words calculated by the algorithm. If we set $\bar{u}_k = (u_1, u_2, \ldots, u_k)$, then we obtain:

$$\lambda(\bar{u}_1) \leq 2 \cdot s(D_0) \qquad \text{(by } (*) \text{ in Lemma 3.6)}$$
$$\lambda(\bar{u}_{k+1}) \leq \lambda(\bar{u}_k) \cdot \lambda(u_{k+1}) \leq 6 \cdot s(D_0) \cdot \lambda(\bar{u}_k)^2$$

From this, we obtain by induction $\lambda(\bar{u}_k) \leq 2^{2^k} \cdot 6^{2^k-1} \cdot s(D_0)^{2^k-1}$. Since $s(D_0) \in \exp(3, O(D_0))$ and $k \leq D_0$, it follows $\lambda(\bar{u}_k) \in \exp(3, O(D_0))$. Hence, each of the Büchi-automata M in line 6 can be constructed in triply-exponential space. Since the recursion depth of the overall algorithm is bounded by the size of the input formula and for each recursive call only a triply exponential amount of information has to be stored, the whole algorithm can be executed in space triply exponential in the size of the input formula. Thus, we proved:

Theorem 3.7. *Let $\mathbb{C} = \{\mathcal{C}_i \mid i \in \mathbb{N}\}$ be a countable set of relations on $\mathbb{N} \cup \{\aleph_0, 2^{\aleph_0}\}$. Let $\mathcal{A}$ be an ω-automatic structure of bounded degree. Then the* FO($\mathbb{C}$)*-theory of $\mathcal{A}$ can be decided in triply exponential space by a Turing machine with oracle $\{(i, \bar{c}) \mid i \in \mathbb{N}, \bar{c} \in \mathcal{C}_i\}$.*

3.3 Expressiveness of the Logic $\mathcal{L}(Q_u)$

Let $\mathcal{A}$ be some structure of bounded degree and let $\varphi(\bar{x})$ be an $\mathcal{L}(Q_u)$-formula with k free variables of quantifier depth d. We want to show that there exists an equivalent first-order formula $\psi(\bar{x})$. For this, we can first extend the signature of $\mathcal{A}$ by the first-order definable relations $\deg_n$ in order to ensure assumptions (2) and (3) from page 330. Now let $\#$ and Φ be the functions from Lemma 3.4 and set

$$I = \{i \mid 1 \le i \le \#(d,k), \mathcal{A} \models \forall \bar{x} : (\Phi(d,k,i) \to \varphi)\}$$

and $\psi = \bigvee_{i \in I} \Phi(d,k,i)$. Then Lemmas 3.3 and 3.4 together with Theorem 3.2 imply $\mathcal{A} \models \forall \bar{x}(\varphi \leftrightarrow \psi)$. This proves:

Corollary 3.8. *Let $\mathcal{A}$ be a τ-structure of bounded degree, and let $\varphi(\bar{x}) \in \mathcal{L}(\mathcal{Q}_u)$. There exists a formula $\psi(\bar{x}) \in \mathrm{FO}$ such that $\mathcal{A} \models \forall \bar{x}(\varphi \leftrightarrow \psi)$.*

The above proof is not effective since it does not give a way to compute the set I effectively. For ω-automatic structures $\mathcal{A}$ of bounded degree, the situation changes since it can be decided in elementary space as to whether $\alpha_i = \forall \bar{x}(\Phi(d,k,i) \to \varphi)$ holds in $\mathcal{A}$:

Corollary 3.9. *Let $\mathbb{C} = \{\mathcal{C}_i \mid i \in \mathbb{N}\}$ be a countable set of relations on $\mathbb{N} \cup \{\aleph_0, 2^{\aleph_0}\}$. Let $\mathcal{A}$ be an ω-automatic structure of bounded degree. For any $\varphi(\bar{x}) \in \mathrm{FO}(\mathbb{C})$, one can construct in elementary space (modulo $\mathbb{C}$) a formula $\psi(\bar{x}) \in \mathrm{FO}$ and a $|\bar{x}|$-dimensional Büchi-automaton M such that for any $\bar{u} \in L^{|\bar{x}|}$:*

$$\mathcal{A} \models \varphi(\bar{u}) \iff \mathcal{A} \models \psi(\bar{u}) \iff \bar{u} \in L_\omega(M) \ .$$

Recall that by Propositions 2.2, 2.5, and 2.7, any relation definable in FO extended by modulo- and cardinality-quantifiers is effectively ω-automatic. A similar statement can be found in Corollary 3.9. Also Theorems 2.8 and 3.7 are similar in as far as they state the decidability of some theories. But the proof strategies are different: while Theorem 2.8 was derived from Propositions 2.2, 2.5, and 2.7, the corresponding statement Theorem 3.7 was used to prove Corollary 3.9.

3.4 Optimality

The main results concerning the powerful logic $\mathcal{L}(Q_u)$ deal with structures satisfying two assumptions: they are ω-automatic and of bounded degree. In this section, we show that the two assumptions we made cannot be relaxed. First, it is shown that relaxing "ω-automatic" to "recursive" makes the results fail:

Theorem 3.10. *There exists a recursive structure $\mathcal{A}$ of bounded degree such that the* FO*-theory of $\mathcal{A}$ is decidable and the* $\mathrm{FO}(\exists^\infty)$*-theory of $\mathcal{A}$ is undecidable.*

Proof. Let $L \subseteq \{0,1\}^*$ be a recursively enumerable, but not recursive set and let M be a Turing machine that, on input of $w \in \{0,1\}^*$, eventually stops if and only if $w \in L$. Let $f(w) \in \mathbb{N} \cup \{\omega\}$ denote the number of steps M performs on input w. The structure $\mathcal{A}$ consists of $f(w)$ many copies of the word $\triangleright w \triangleleft$ for

any $w \in \{0,1\}^*$ (seen as labeled finite successor structures), i.e., $\mathcal{A}$ is a labeled directed graph whose degree is bounded by 2. Then in $\mathrm{FO}(\exists^\infty)$, we can express that M does not stop on input w, hence this theory is undecidable. Gaifman's theorem, on the other hand, yields that the FO-theory is decidable. □

By choosing a more complicated but still recursive counting quantifier, we can show that Theorem 3.7 even fails for locally finite automatic structures.

Theorem 3.11. *There is a recursive set $K \subseteq \mathbb{N}$ and a locally finite automatic structure $\mathcal{A}$ such that the* $\mathrm{FO}(C_K)$*-theory of $\mathcal{A}$ is undecidable.*

Proof. We start with the structure $(\mathbb{N}, +1)$ and attach, to any element $n \in \mathbb{N}$, additional n nodes via a relation t. The resulting structure $\mathcal{A}$ is automatic. Let $a_1, a_2, a_3, \ldots$ be a recursive enumeration of the non-recursive set $A \subseteq \mathbb{N}$ and let K denote the recursive set $\{a_1 + \cdots + a_i \mid i \geq 1\}$. Let $\varphi_K(x)$ be the formula $C_K y : t(x,y)$. Then $m \in A$ if and only if there exists y satisfying $\varphi_K(y) \wedge \varphi_K(y+m) \wedge \bigwedge_{1 \leq k < m} \neg\varphi_K(y+k)$. □

4 An Open Problem

In view of Theorems 2.8 and 3.11 it might be an interesting problem to characterize those subsets $K \subseteq \mathbb{N}$ such that for every (ω-)automatic structure (not necessarily of bounded degree), the $\mathrm{FO}(C_K)$-theory of $\mathcal{A}$ is decidable. Note that by Theorem 2.8, this is true for every semi-linear set K. Since $(\mathbb{N}, \leq)$ is automatic and since $x \in K$ can be expressed as $C_K y : y < x$, the set K has to be decidable. As observed by an of the referees, even the monadic second order theory of $(\mathbb{N}, \leq, K)$ has to be decidable since it can be reduced to the $\mathrm{FO}(C_K)$-theory of the ω-automatic structure $(\{0,1\}^\infty, \leq)$. Furthermore, K cannot be the range of any non-linear polynomial over $\mathbb{N}$ [5] nor can it be k-recognizable (for some k) but not semi-linear [2].

References

1. M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin. A model-theoretic approach to regular string relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'2001)*, pages 431–440. IEEE Computer Society Press, 2001.
2. A. Bès. Undecidable extensions of Büchi arithmetic and Cobham-Semenov theorem *Journal of Symbolic Logic*, 62(4):1280–1296, 1997.
3. A. Blumensath. Automatic structures. Diploma thesis, RWTH Aachen, 1999.
4. A. Blumensath and E. Grädel. Finite presentations of infinite structures: Automata and interpretations. *Theory of Computing Systems*, 37(6):641–674, 2004.
5. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.
6. D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. F. Levy, M. S. Paterson, and W. P. Thurston. *Word processing in groups.* Jones and Bartlett, Boston, 1992.
7. W. Hodges. *Model Theory.* Cambridge University Press, 1993.

8. B. R. Hodgson. On direct products of automaton decidable theories. *Theoretical Computer Science*, 19:331–335, 1982.
9. H. Ishihara, B. Khoussainov, and S. Rubin. Some results on automatic structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'2002)*, pages 235–244. IEEE Computer Society Press, 2002.
10. H. J. Keisler and W. B. Lotfallah. A local normal form theorem for infinitary logic with unary quantifiers. *Mathematical Logic Quarterly*, 51(2):137–144, 2005.
11. B. Khoussainov and A. Nerode. Automatic presentations of structures. In *LCC: International Workshop on Logic and Computational Complexity*, number 960 in Lecture Notes in Computer Science, pages 367–392, 1994.
12. B. Khoussainov, S. Rubin, and F. Stephan. Automatic partial orders. *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS'2003)*, pages 168-177. IEEE Computer Society Press, 2003.
13. B. Khoussainov, S. Rubin, and F. Stephan. Definability and regularity in automatic structures. In V. Diekert and M. Habib, editors, *Proceedings of the 21th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2004), Montpellier (France)*, number 2996 in Lecture Notes in Computer Science, pages 440–451. Springer, 2004.
14. D. Kuske. Is Cantor's theorem automatic. In *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2003), Almaty (Kazakhstan)*, number 2850 in Lecture Notes in Artificial Intelligence, pages 332–345, 2003.
15. D. Kuske and M. Lohrey. First-order and counting theories of ω-automatic structures. Technical Report 2005-07, Universität Stuttgart, 2005. available at ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2005-07
16. M. Lohrey. Automatic structures of bounded degree. In *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2003), Almaty (Kazakhstan)*, number 2850 in Lecture Notes in Artificial Intelligence, pages 346–360, 2003.
17. D. Perrin and J.-E. Pin. *Infinite Words.* Pure and Applied Mathematics vol. 141. Elsevier, 2004.
18. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

Parity Games Played on Transition Graphs of One-Counter Processes*

Olivier Serre

LIAFA, Université Paris VII & CNRS

Abstract. We consider parity games played on special pushdown graphs, namely those generated by one-counter processes. For parity games on pushdown graphs, it is known from [23] that deciding the winner is an ExpTime-complete problem. An important corollary of this result is that the μ-calculus model checking problem for pushdown processes is ExpTime-complete. As one-counter processes are special cases of pushdown processes, it follows that deciding the winner in a parity game played on the transition graph of a one-counter process can be achieved in ExpTime. Nevertheless the proof for the ExpTime-hardness lower bound of [23] cannot be adapted to that case. Therefore, a natural question is whether the ExpTime upper bound can be improved in this special case. In this paper, we adapt techniques from [11, 4] and provide a PSpace upper bound and a DP-hard lower bound for this problem. We also give two important consequences of this result. First, we improve the best upper bound known for model-checking one-counter processes against μ-calculus. Second, we show how these games can be used to solve pushdown games with winning conditions that are Boolean combinations of a parity condition on the control states with conditions on the stack height.

1 Introduction

Infinite two-player games with perfect information allow us to encode several challenging problems from formal verification, and this is one of the reasons why they are so intensively studied for several years. Several model-checking problems can be expressed as decision problems for games: the most famous example is that the μ-calculus model-checking problem is polynomially equivalent to the solution of a parity game. This correspondence was first proved for finite graphs [6] and later extended to various classes of infinite graphs, *e.g.* pushdown graphs [23, 24]. Two-player games also offer a very convenient framework to represent interaction of a program with some (possibly hostile) environment. In this approach, the first player represents the program while the second player simulates the environment. A winning strategy expresses a property that must hold *whatever* the environment does. Hence, finding a winning strategy for the first player

* This research has been partially supported by the European Community Research Training Network "Games and Automata for Synthesis and Validation" (GAMES), (contract HPRN-CT-2002-00283), see www.games.rwth-aachen.de.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 337–351, 2006.
© Springer-Verlag Berlin Heidelberg 2006

allows to synthesize a controller that restricts the program and ensures that the property expressed by the winning condition always holds [2].

The most standard setting of verification and synthesis only considers finite games. Nevertheless infinite models arise when recursive programs or programs with variable on infinite domains are considered. Therefore, solving games on such infinite objects is a natural question. The special case of pushdown processes, have been intensively studied from the games point of view (see *e.g.* [23, 11, 5, 3, 7, 19]) and the most important consequence for model-checking is that for pushdown processes, the μ-calculus model-checking problem is ExpTime-complete [23].

In this paper, we consider a natural subclass of pushdown processes, namely one-counter processes with zero test. Verification problems for one-counter processes have intensively been studied (see *e.g.* [9, 8]) but, for model-checking problems, most of the complexity results concern lower bounds whereas upper bounds generally follow from known results for pushdown processes. Hence, this frequently yields complexity gaps, as for μ-calculus model-checking where the lower bound is DP-hard [8] whereas the upper bound is ExpTime [23].

We consider parity games played on one-counter graphs and provide a PSpace algorithm to decide the winner in such games. Our procedure relies on a tricky adaptation and a precise analysis of the techniques from [11, 4] that were originally developed for pushdown games. Our result improves the ExpTime upper bound inherited from pushdown games [23]. As a by-product, it improves the best known upper bound for the μ-calculus model-checking for one-counter processes from ExpTime to PSpace.

Another consequence of our main result concerns pushdown games equipped with winning conditions that combine both regular conditions and conditions on how the stack height evolves during a play (*e.g.* unboundedness). Special cases of these games have been studied and shown to be decidable [5, 3, 7]. Here, we capture a larger class of games and we provide a more intuitive construction which generalizes those for parity games [23] and for strict unboundedness [19]. Moreover, our construction is very general, and one does not need to provide a specific construction/proof for each possible kind of winning condition neither to prove preliminary results on the existence of memoryless strategy.

The paper is organized as follows. In Section 2, we give the main definitions. Section 3 provides a PSpace algorithm to solve one-counter parity games and a DP lower-bound is presented for one-counter reachability games. The consequences of these results are presented in the two last sections: Section 4 considers the μ-calculus model-checking problem while Section 5 focuses on pushdown games. Due to the page limit, missing proofs and extra details can be found in [20].

2 Definitions

An *alphabet* A is a finite set of letters. A^* denotes the set of *finite words* on A and A^ω the set of *infinite words* on A. The empty word is denoted by ε.

Infinite two-player games. Let $G = (V, E)$ denote a (possibly infinite) graph with vertices V and edges $E \subseteq V \times V$. Let $V_{\mathbf{E}} \cup V_{\mathbf{A}}$ be a partition of V between two players Eve and Adam. A *game graph* is such a tuple $\mathcal{G} = (V_{\mathbf{E}}, V_{\mathbf{A}}, E)$. An *infinite two-player game* on a game graph $\mathcal{G}$ is a pair $\mathbb{G} = (\mathcal{G}, \Omega)$, where $\Omega \subseteq V^\omega$ is a *winning condition.*

The players, Eve and Adam, play in $\mathbb{G}$ by moving a token between vertices. A *play* from some vertex v_0 proceeds as follows: the player owning v_0 chooses a successor v_1 such that $(v_0, v_1) \in E$. Then the player owning v_1 chooses a successor v_2 and so on, forever. If at some point one of the players cannot move, she/he looses the play. Otherwise, the play is an infinite word $\Lambda \in V^\omega$ and is won by Eve if and only if $\Lambda \in \Omega$. As one can always add loops on dead-end vertices, and slightly modify the winning condition to make looping plays on some dead-end vertex loosing for the player that controls it, we will assume in the sequel that all plays are infinite. A *partial play* is any prefix of a play.

A strategy for Eve is a function assigning, to any partial play ending in some vertex $v \in V_{\mathbf{E}}$, a vertex v' such that $(v, v') \in E$. Eve *respects a strategy* Φ during some play $\Lambda = v_0 v_1 v_2 \cdots$ if $v_{i+1} = \Phi(v_0 \cdots v_i)$, for all $i \geq 0$ such that $v_i \in V_{\mathbf{E}}$. Finally a strategy for Eve is *winning* from some position $v \in V$, if any play starting from v, where Eve respects Φ, is won by her. A vertex $v \in V$ is winning for Eve if she has a winning strategy from v. Symmetrically, one defines strategies and winning positions for Adam.

A game $\mathbb{G}$ is *determined* if, from any position, either Eve or Adam has a winning strategy. For all games considered in this article one can use Martin's Theorem [15] and conclude that they are determined.

For more details and basic results on games, we refer to [21, 26].

Pushdown games. *Pushdown processes* provide a natural model for programs with recursive procedures. They are like nondeterministic pushdown automata except that they have no input (and therefore no initial state neither final state).

More formally, a *pushdown process* is a tuple $\mathcal{P} = \langle Q, \Gamma, \bot, \Delta \rangle$ where Q is a finite set of states, Γ is a finite stack alphabet that contains a special bottom-of-stack symbol $\bot$ and $\Delta : Q \times \Gamma \rightarrow 2^{(\{skip(q), pop(q), push(q,\gamma) \mid q \in Q, \gamma \in \Gamma \setminus \{\bot\}\})}$ is the transition relation. We additionally require that, for all $q \in Q$, $\Delta(q, \bot)$ does not contain any element of the form $pop(q')$.

A *stack* is any word in $St = (\Gamma \setminus \{\bot\})^* \cdot \bot$. A configuration of $\mathcal{P}$ is a pair (q, σ) with $q \in Q$ and $\sigma \in St$. Note that the top stack symbol in some configuration (q, σ) is the leftmost symbol of σ.

Any pushdown process $\mathcal{P}$ induces an infinite graph, called *pushdown graph*, denoted $G = (V, E)$, whose vertices are the configurations of $\mathcal{P}$, and edges are defined by the transition relation Δ, i.e., from a vertex $(p, \gamma\sigma)$ one has edges to:

- $(q, \gamma\sigma)$ whenever $skip(q) \in \Delta(p, \gamma)$.
- (q, σ) whenever $pop(q) \in \Delta(p, \gamma)$.
- $(q, \gamma'\gamma\sigma)$ whenever $push(q, \gamma') \in \Delta(p, \gamma)$.

Consider a partition $Q_{\mathbf{E}} \cup Q_{\mathbf{A}}$ of Q between Eve and Adam. It induces a natural partition $V_{\mathbf{E}} \cup V_{\mathbf{A}}$ of V by setting $V_{\mathbf{E}} = Q_{\mathbf{E}} \times St$ and $V_{\mathbf{A}} = Q_{\mathbf{A}} \times St$. The

resulting game graph $\mathcal{G} = (V_{\mathbf{E}}, V_{\mathbf{A}}, E)$ is called a *pushdown game graph*. Finally, a *pushdown game* is a game played on such a game graph.

The *regular winning conditions* on pushdown games are inherited from the standard acceptance condition for automata on infinite words. The simplest one is the *reachability condition*. Let $F \subseteq Q$ be a set of *final states*, and let V_F be the set of configuration which control state is in F. The reachability condition is the winning condition defined by $\Omega_{reach}(F) = \{v_0 v_1 \cdots \mid \exists v_i \in V_F\}$. Using the notion of final states one can define the *Büchi* condition and its dual winning condition the *co-Büchi conditions*: $\Omega_{Buc}(F) = \{v_0 v_1 \cdots \mid \forall i \geq 0\ \exists j \geq i \text{ s.t. } v_i \in V_F\}$ and $\Omega_{co-Buc}(F) = V^\omega \setminus \Omega_{Buc}(F)$.

Let *col* be a coloring function from Q into a finite set of colors $C \subset \mathbb{N}$. This function is easily extended into a function from V into C by setting $col((q,\sigma)) = col(q)$. The parity condition is the winning condition defined by:

$$\Omega_{par} = \{v_0 v_1 \cdots \mid \liminf((col(v_i))_{i\geq 0}) \text{ is even}\} .$$

For a parity game played on a pushdown graph, the main question is to decide which player has a winning strategy from some given configuration. It is easily seen that this last question can be reduced to decide the winner for configurations with empty stack. For the general case of parity games played on pushdown graph, this last problem has been fully characterized by Walukiewicz [23].

Theorem 1. *[23] Deciding the winner from some configuration of empty stack in a pushdown parity game is an* ExpTime *problem. Moreover, this problem is* ExpTime*-hard even if the winning condition is a reachability one.*

One-counter games. A *one-counter process* is a special case of a pushdown process $\mathcal{P} = \langle Q, \Gamma, \perp, \Delta\rangle$ where $\Gamma = \{1, \perp\}$ consists of a single stack symbol 1 together with the bottom-of-stack symbol. It therefore corresponds to a finite state machine equipped with a counter that can be test to zero. The notions of *one-counter graphs*, *one-counter game graphs* and *one-counter games* are induced by the one for pushdown processes.

Remark 1. Note that our model of one-counter processes can test whether the stack (equivalently the counter value) is empty (equivalently equals 0). This follows from the fact that in the definition of pushdown processes the bottom-of-stack belongs to the stack alphabet, and hence can be checked as top stack symbol when performing an action.

Theorem 1 implies an ExpTime upper bound to decide the winner in a one-counter parity game. The ExpTime-hard lower bound of Theorem 1 is established by coding the computation of an alternating Turing machine using linear space into a reachability pushdown game. The main idea of the reduction is that the pushdown process is built so that its stack is a description of the prefix of a computation of the Turing machine. Therefore this construction strongly relies on the fact that the stack alphabet is large enough to describe configurations of the Turing machine. Hence this proof cannot be adapted to the case of one-counter game.

A natural question is thus to check whether it is possible, in the special case of one-counter games, to improve the ExpTime upper bound. We positively answer this question in Theorem 2 by providing a PSpace algorithm.

3 Deciding the Winner in a One-Counter Parity Game

3.1 Upper Bound

Intuition. In [11], Kupferman and Vardi have proposed a new approach, based on automata, to solve model-checking problem for pushdown graphs. The main idea was to reduce a model-checking problem to an emptiness problem for a class of tree automata, namely alternating two-way parity tree automata. This technique can then be adapted to solve parity pushdown games [4].

Let us first informally recall the construction of [4], and explain how to simplify it in the special case of one-counter parity games. It is rather standard to consider that the complete infinite tree of arity k is a representation of the set of all finite words on an alphabet of cardinality k. Each node in this tree is labeled by the last letter of the word it represents: hence the word associated to some node is obtained by considering the sequence of labels of the nodes on the path from the root to the current one. Using this fact, a play in a pushdown game can be considered as an (infinite) path in such a tree in which a node encodes the stack content while an extra information describes the control state. As there are finitely many control states, and as the possible moves only depend on the control state and on the top stack symbol (that is the label of the current node), this representation of a play can be seen as a path in the run of some tree automaton on the complete infinite tree of arity k, where k is the size of the pushdown stack alphabet without the bottom-of-stack symbol. This tree automaton can go in both directions in the tree: it goes down to simulate a rule that pushes some new symbol, it goes up to simulate a popping rule and it stay in the same node to simulates a skip rule. For each possible move, the control state has to be updated in accordance with the pushdown transition rules. As we aim to simulate a game, the tree automaton needs to be alternating: existential states are those associated to Eve's states while universal states are those associated to Adam's states. Finally, the acceptance condition is inherited from the winning condition, and is therefore a parity condition. The complete infinite tree of arity k is accepted if and only if Eve has a winning strategy in the pushdown game.

The previous tree automaton works on the complete infinite tree and the arity of this tree is the cardinality of the stack alphabet without the bottom-of-stack symbol. Hence, if we restrict ourselves to one-counter processes instead of general pushdown processes, the arity is equal to 1 and instead of a tree we have to consider a simpler model, namely the infinite word $\perp 1^\omega$. Therefore, it follows that to decide the winner in a one-counter parity game it is sufficient to check emptiness for an alternating two-way parity *word* automaton. In Proposition 2 we will show that emptiness for these word automata can be checked in PSpace and hence it will yield a PSpace procedure to decide the winner in a one-counter parity game (Theorem 2).

Definitions. Given a set S of variables, we denote by $\mathcal{B}^+(S)$ the set of positive boolean formulas over S with *true* and *false*. A subset $S' \subseteq S$ *satisfies* a formula in $\mathcal{B}^+(S)$ if this formula is satisfied by the valuation assigning true to every variable in S' and false to every variable in $S \setminus S'$.

An *alternating two-way parity word automaton* $\mathcal{A}$ is a tuple $\langle Q, A, q_{\text{in}}, \delta, col\rangle$, where Q is a finite set of control states, A is a finite input alphabet, $q_{\text{in}} \in Q$ is an initial state, δ is a mapping from $Q \times A$ to $\mathcal{B}^+(Q \times \{-1, 0, 1\})$, and *col* is a mapping from Q to a finite set of colors $C \subset \mathbb{N}$. An *alternating one-way parity word automaton* corresponds to the special case where $\delta : Q \times A \to \mathcal{B}^+(Q \times \{1\})$.

A *run* of $\mathcal{A}$ on an infinite word $u = a_0 a_1 \cdots \in A^\omega$ is an infinite $(Q \times \mathbb{N})$-labeled tree such that its root is labeled by $(q_{\text{in}}, 0)$, and for every vertex x labeled by some (q, n) with sons labeled by $(q_1, n_1), \ldots, (q_k, n_k)$, the set $\{(q_1, n_1 - n), \ldots, (q_k, n_k - n)\} \subset Q \times \{-1, 0, 1\}$ satisfies $\delta(q, a_n)$. A run is *accepting* if and only if for every infinite branch, the smallest infinitely repeated color is even, where the color of a node labeled by some (q, n) is $col(q)$. Finally, an infinite word is *accepted* if there exists an accepting run on it, and we denote by $L(\mathcal{A})$ the set of all words accepted by $\mathcal{A}$.

The construction. Let $\mathcal{C} = \langle Q, \{1, \bot\}, \bot, \Delta\rangle$ be a one-counter process equipped with a partition $Q_{\mathbf{E}} \cup Q_{\mathbf{A}}$ of its control states, and with a coloring function $col : Q \to C$. Let $\mathbb{G}$ be the one-counter parity game induced by the preceding partition and the coloring function *col*. Let q_{in} be some state in Q. We are interesting in deciding whether $(q_{\text{in}}, \bot)$ is winning for Eve in $\mathbb{G}$.

To solve this problem, instead of using the techniques from [23], that would lead to an ExpTime procedure, we adapt the techniques developed in [11, 4], and note that it reduces our problem to the emptiness problem for alternating two-way parity word automaton.

Let us consider the alternating two-way parity word automaton $\mathcal{A} = \langle Q, \{1, \bot\}, q_{\text{in}}, \delta, col\}$ where the transition function δ is defined by:

- for every $q \in Q_{\mathbf{E}}$ and for every $a \in \{1, \bot\}$, $\delta(q, a)$ equals $[\bigvee_{push(q',1) \in \Delta(q,a)} (q', 1)] \vee [\bigvee_{skip(q') \in \Delta(q,a)} (q', 0)] \vee [\bigvee_{pop(q') \in \Delta(q,a)} (q', -1)]$.
- for every $q \in Q_{\mathbf{A}}$ and for every $a \in \{1, \bot\}$, $\delta(q, a)$ equals $[\bigwedge_{push(q',1) \in \Delta(q,a)} (q', 1)] \wedge [\bigwedge_{skip(q') \in \Delta(q,a)} (q', 0)] \wedge [\bigwedge_{pop(q') \in \Delta(q,a)} (q', -1)]$.

We have the following straightforward proposition.

Proposition 1. *[11, 4] The configuration $(q_{in}, \bot)$ is winning for Eve in $\mathbb{G}$ if and only if $\mathcal{A}$ accepts the infinite word $\bot 1^\omega$.*

Checking whether $\mathcal{A}$ accepts the word $\bot 1^\omega$ is closely related to checking emptiness for a language accepted by an alternating two-way parity word automaton. This problem was studied by Vardi in [22] in the more general setting of two-way alternating parity tree automata, and then this construction was adapted for alternating two-way *Büchi* word automata in [17, 10]. In the case of tree automata, checking emptiness is in ExpTime, while in the case of Büchi word automata the problem is in PSpace. The following proposition, extends this last result to the case of parity acceptance condition.

Proposition 2. *Deciding emptiness for a language accepted by an alternating two-way parity word automaton can be achieved in* PSPACE.

Proof. We only give the main ideas and explain how the construction of [17, 10] is extended to our setting.

Let $\mathcal{A}$ be an alternating two-way parity word automaton. The first step is to build an alternating one-way parity automaton $\mathcal{B}$ such that $L(\mathcal{B}) \neq \emptyset$ if and only if $L(\mathcal{A}) \neq \emptyset$. Moreover the number of control states of $\mathcal{B}$ is polynomial in the number of control states of $\mathcal{A}$. However the size of its alphabet is exponential but note that it is not important for the complexity of emptiness checking.The construction of $\mathcal{B}$ directly follows from the ones in [22, 17, 10]. A precise analysis of the structure of $\mathcal{B}$ is given by the following lemma.

Lemma 1. *Let $\mathcal{A} = \langle Q, A, q_{in}, \delta, col \rangle$ be an alternating two-way parity word automaton, let $n = |Q|$ and let d be the number of colors involved in the parity condition. Then there exists an alternating one-way parity word automaton $\mathcal{B}$ such that $L(\mathcal{B}) \neq \emptyset$ if and only if $L(\mathcal{A}) \neq \emptyset$. Moreover, $L(\mathcal{B}) = L(\mathcal{B}_1) \cap L(\mathcal{B}_2)$, where $\mathcal{B}_1$ is an alternating one-way automaton without acceptance condition (every run, when exists, is accepting) and has $\mathcal{O}(nd)$ states, and $\mathcal{B}_2$ is a purely universal (its transition function takes value into the boolean formulas made only of conjunctions) one-way parity automaton (with d colors) and has $\mathcal{O}(nd)$ states.*

Let n_1 and n_2 be the respective sizes of the set of control states of $\mathcal{B}_1$ and $\mathcal{B}_2$. As $\mathcal{B}_2$ is purely universal, its dual automaton $\overline{\mathcal{B}_2}$ is a non deterministic parity automaton using d colors and having $\mathcal{O}(n_2)$ states. It is then standard to build a nondeterministic Büchi automaton $\overline{\mathcal{B}'_2}$ that recognizes the same language than $\overline{\mathcal{B}_2}$ (that is the complement of $L(\mathcal{B}_2)$) and having $\mathcal{O}(n_2 d)$ states (see [13] for instance). Dualizing $\overline{\mathcal{B}'_2}$ yields a purely universal co-Büchi automaton $\mathcal{B}'_2$ with $\mathcal{O}(n_2 d)$ states and such that $L(\mathcal{B}'_2) = L(\mathcal{B}_2)$.

Now, the *intersection* of $\mathcal{B}_1$ and $\mathcal{B}'_2$ provides an alternating co-Büchi automaton $\mathcal{B}'$ with $\mathcal{O}(n_2 d + n_1) = \mathcal{O}(n^2 d)$ states that recognizes the language $L(\mathcal{B}_1) \cap L(\mathcal{B}'_2) = L(\mathcal{B}_1) \cap L(\mathcal{B}_2) = L(\mathcal{B})$. As checking emptiness for an alternating co-Büchi automaton can be achieved in PSPACE (see [12] for instance), we conclude that one can check whether $L(\mathcal{A})$ is empty in PSPACE. □

Propositions 1 and 2 directly imply the following theorem.

Theorem 2. *Deciding the winner in a one-counter parity game can be done in* PSPACE.

3.2 Lower Bound

In this section, we give a lower bound for the problem of deciding the winner in a one-counter reachability game. Due to the symmetry of the problem, the lower bound should be robust under complementation: we provide such a lower bound, namely DP-hardness. Note that DP-hardness is a rather standard lower

bound for problems related to one-counter process, *e.g.* the *EF* model-checking problem for one-counter processes [8].

A language L is in the complexity class DP if and only if there are two languages $L_1 \in$ NP and $L_2 \in$ co-NP such that $L = L_1 \cap L_2$.

The SAT-UNSAT problem is the following one: given two Boolean formulas ψ_1 and ψ_2, both in conjunctive normal form with three literals per clause, decide whether φ_1 is satisfiable and φ_2 is not. It is rather immediate to prove that SAT-UNSAT is DP-complete [16].

Let us first explain how to polynomially reduce 3-SAT to decide the winner in a one-counter reachability game. Let $X = \{x_1, \dots, x_k\}$ be a set of variables and let ψ be some Boolean formula in conjunctive normal form with 3 literals per clause. Let denote $\psi = C_1 \wedge C_2 \wedge \cdots \wedge C_h$, where $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$ for all $i = 1, \dots, h$ with $l_{i,j} \in \{x, \overline{x} \mid x \in X\}$, for $j = 1, 2, 3$.

For every $i \geq 1$, let ρ_i denote the i-th prime number. A valuation of X is a mapping from X into $\{0, 1\}$, that is a tuple in $\{0, 1\}^k$. Let $\tau : \mathbb{N} \to \{0, 1\}^k$ be the function defined by $\tau(n) = (b_1, b_2, \dots, b_k)$ where $b_j = 0$ if $n = 0 \mod \rho_j$ and $b_j = 1$ otherwise. The Chinese remainder lemma implies that τ is surjective.

Consider now the following informal game. Eve chooses some integer n encoding a valuation that she claims to satisfy ψ. Then Adam picks a clause C_i that he claims not to be satisfied by the preceding valuation. Eve contests by giving a literal of C_i that she claims to be evaluated to true by the preceding valuation. Finally Adam checks whether this literal is evaluated to true: if it is the case, then Eve wins, otherwise Adam does. It is then easily seen that Eve has a winning strategy if and only if ψ is satisfiable.

This game can be encoded into a one-counter reachability game. For the first step, Eve increments the counter until it equals n. For the second step, Adam indicates the clause by changing the control state. In the third step, Eve indicates the literal by changing the control state. Finally, Adam check whether the literal evaluates to true by decrementing the counter while performing a modulo ρ_k counting, where the literal was x_k or $\overline{x_k}$.

Now, if one wants to reduce SAT-UNSAT, it suffices to add a preliminary step to the previous game. Let (ψ_1, ψ_2) be the instance of SAT-UNSAT. First Adam picks ψ_1 or ψ_2. In the first case Eve and Adam play the previous game. In the second case, they play the dual game where Adam is now the one that has to provide a valuation for ψ_2, and where Eve wins if and only if ψ_2 is not satisfiable. Eve wins the main game if and only if she can win both sub-games, that is if and only if ψ_1 is satisfiable while ψ_2 is not. Hence, we have the following result.

Theorem 3. *Deciding the winner in a one-counter reachability game is a* DP-*hard problem.*

Remark 2. An alternative proof for this result is the following one: consider the *EF* model-checking problem for one-counter automata. In [8] this problem is shown to be DP-hard. One can then easily reduce it to decide the winner in a one-counter reachability game. Nevertheless, we think that the proof we gave for Theorem 3 is more intuitive and better here as it is self-contained.

4 Model-Checking Propositional μ-Calculus Against One-Counter Trees

In this section we rephrase Theorem 3 in the framework of propositional μ-calculus model-checking problem for one-counter trees. An important consequence is that it improves from ExpTime to PSpace the best complexity bound known for this problem.

Propositional μ-calculus is a very powerful fix point logic that allows to specify a large class of properties of (non-terminating) systems. Moreover, many important temporal logic were shown to be fragments of μ-calculus. For definitions and results on μ-calculus, we refer to [1].

Models of μ-calculus formulas are transitions systems, that is graphs equipped with functions that assign to any propositional constant the set of vertices where it holds. The μ-calculus model-checking problem is to decide, for a given model $\mathcal{M}$, a state s of $\mathcal{M}$, and a μ-calculus formula φ, whether φ holds in s. In the sequel we are interested in the special case where $\mathcal{M}$ is the unfolding of a one-counter graph, called a *one-counter tree.*

A *standard* technique to solve a μ-calculus model-checking problem is to construct a parity game in which Eve has a winning strategy if and only if the model satisfies the formula. The game graph is obtained by considering the synchronized product of a finite game graph, representing the formula φ, with the model $\mathcal{M}$. This idea was first used in [6] for finite transition systems, and was then adapted in [23] for pushdown trees (see also [25] for a general presentation of the technique). In the case of pushdown trees, an important point to note is that in the synchronized product, the stack alphabet remains unchanged (the product is done in the control states). Hence, using the same construction for one-counter trees reduces the μ-calculus model-checking problem for a one-counter tree to solve a one-counter parity game. Conversely, it follows from [23] that solving a one-counter parity game reduces to a μ-calculus model checking problem. As both reductions are polynomial, we obtain the following consequence of Theorem 2.

Theorem 4. *The propositional μ-calculus model-checking problem for one-counter trees can be solved in* PSpace *and is* DP*-hard.*

Remark 3. Note that the DP-hardness was already known, as it is a consequence of the DP-hardness of the model-checking problem for the branching-time temporal logic *EF* [8] which is a fragment of the propositional μ-calculus.

5 Application to Pushdown Games

In section 2 we have defined the regular winning conditions. Nevertheless, when considering pushdown games, non-regular winning conditions arise naturally. In particular, one can require conditions on how the stack height evolves during the play. For some configuration $v = (q, \sigma\perp)$ in a pushdown graph, let $sh(v) = |\sigma|$ denote the stack height in v. The *unboundedness condition* requires that the

stack height is not bounded. Its dual condition is the *boundedness condition*. Both conditions are formally defined as follows:

- $\Omega_{Ubd} = \{v_0v_1\cdots \mid \limsup((sh(v_i)_{i\geq 0}) = \omega\}$.
- $\Omega_{Bd} = \{v_0v_1\cdots \mid \exists B \geq 0 \text{ s.t. } sh(v_i) < B\ \forall i \geq 0\}$.

If we replace the lim sup by a lim in the definition of the unboundedness condition then we obtain the *strict unboundedness condition* which enforces the stack height to converge to infinity. Its dual version, the *repeating condition* requires that some stack height (equivalently, some vertex) is infinitely often visited. Both conditions are formally defined as follows:

- $\Omega_{StUbd} = \{v_0v_1\cdots \mid \lim((sh(v_i))_{i\geq 0}) = \omega\}$.
- $\Omega_{Rep} = \{v_0v_1\cdots \mid \exists B \geq 0 \text{ s.t. } \forall j \geq 0\ \exists i \geq j \text{ s.t. } sh(v_i) = B\}$.

These four winning conditions will be designated as *stack conditions*. Pushdown games with stack conditions are known to be decidable in ExpTime [5, 19, 3, 7, 18]. In the sequel we consider winning conditions that are a Boolean combination of stack conditions with a parity condition. For instance the winning condition $\Omega_{par} \cap \Omega_{Ubd} \cap \Omega_{Rep}$ requires that the smallest infinitely visited color has to be even and that arbitrary large stack height occurs while some level is infinitely repeated. Note that the winning condition $\Omega_{Ubd} \cap \Omega_{Rep}$ was already mentioned in [5] and can be rephrased as: there exists infinitely many vertices that are infinitely often visited during the play. Decidability of pushdown games with this winning condition was open and is a consequence of the main result of this section.

Games equipped with winning conditions that are a Boolean combination of a parity condition and of an unboundedness condition have been shown to be decidable in [3] when restricting to Büchi conditions and in [7] for the general case. For all these games an ExpTime-complete complexity bound has been provided.

The main result of this section is to provide an ExpSpace procedure to solve these games and more generally to solve the ones equipped with a Boolean combination of a parity condition and of stack conditions. Even if the complexity bound may not be optimal here, the results are more general and the presentation and proof techniques are much simpler and unified. Indeed, the construction is a generalization of the one for parity condition, and it *separates* all conditions involved in the Boolean combination, which allows to reason independently on these conditions and leads to a very flexible construction. Moreover, no preliminary result on memoryless strategy is needed, while it was the case in [7].

From now on, we fix a pushdown process $\mathcal{P} = \langle Q, \Gamma, \bot, \Delta\rangle$, a partition $Q_{\mathbf{E}} \cup Q_{\mathbf{A}}$ of its control states and a coloring function $col : Q \rightarrow \{0, \ldots, d\}$. Let $\mathcal{G}$ be the corresponding game graph, and let Ω_{par} be the parity condition induced by col.

For an infinite play $\Lambda = v_0v_1\cdots$, let $Steps_\Lambda$ be the set of indices of positions where no configuration of strictly smaller stack height is visited later in the play. More formally, $Steps_\Lambda = \{i \in \mathbb{N} \mid \forall j \geq i\ sh(v_j) \geq sh(v_i)\}$. Note that $Steps_\Lambda$ is always infinite and hence induces a factorization of the play Λ into finite pieces.

For all pair $(i,j) \in Steps_\Lambda$, with $i \neq j$ and such that there is no $k \in Steps_\Lambda$ such that $i < k < j$, we define $mcol(i,j) = \min\{col(v_k) \mid i \leq k \leq j\}$ and

$$kind(i,j) = \begin{cases} S & \text{if } sh(v_j) = sh(v_i) + 1 \\ (B,h) & \text{if } sh(v_j) = sh(v_i) \text{ and } h = \max\{sh(v_k) - sh(v_i) \mid i \leq k \leq j\} \end{cases}$$

In the factorization induced by $Steps_\Lambda$, a factor $v_i \cdots v_j$ will be called a *bump* of height h if $kind(i,j) = (B,h)$, and will be called a *Stair* if $kind(i,j) = S$.

For any play Λ with $Steps_\Lambda = \{n_0 < n_1 < \cdots\}$, one can define two sequences $(mcol_i^\Lambda)_{i\geq 0} \in \mathbb{N}^\mathbb{N}$ and $(kind_i^\Lambda)_{i\geq 0} \in (\{S\} \cup (\{B\} \times \mathbb{N}))^\mathbb{N}$ defined by $mcol_i^\Lambda = mcol(n_i, n_{i+1})$ and $kind_i^\Lambda = kind(n_i, n_{i+1})$.

These sequences fully characterize the parity conditions and the stack conditions.

Proposition 3. *For a play Λ the following equivalences hold*

1. *$\Lambda \in \Omega_{par}$ iff $\liminf((mcol_i^\Lambda)_{i\geq 0})$ is even.*
2. *$\Lambda \in \Omega_{Ubd}$ iff either $\{kind_i^\Lambda \mid i \geq 0\}$ contains (B,h) for any $h \geq 0$, or S appears infinitely often in $(kind_i^\Lambda)_{i\geq 0}$.*
3. *$\Lambda \in \Omega_{StUbd}$ iff S appears infinitely often in $(kind_i^\Lambda)_{i\geq 0}$.*

By dualization, one obtains similar characterizations for Ω_{Bd} and Ω_{Rep}.

The main idea used in [23] to solve parity pushdown game is to build a parity game played on an exponentially larger finite graph with the same number of colors. This new game *simulates* the pushdown game, in the sense that the sequences of visited colors during a correct simulation play are exactly the sequences $(mcol_i^\Lambda)_{i\geq 0}$ for plays Λ in the original pushdown game. Moreover, a play in which a player does not correctly simulate the pushdown game is loosing for that player. From this construction follows the ExpTime upper bound.

Let us explain how to extend this technique to handle stack conditions. When considering the strict unboundedness condition, it is sufficient to detect in the simulation game of [23] whether the currently simulated factor is a stair or a bump. Therefore, this construction can be easily adapted to reduce a pushdown games with a strict unboundedness winning condition to a Büchi game played on a finite game graph (the Büchi condition enforcing to simulate an infinite number of stairs) [19, 18]. Nevertheless, for bumps, one cannot express any property on their height.

Consider the unboundedness condition. A play satisfies it either if it satisfies the strict unboundedness condition (which can be encoded by a Büchi condition) or if some stack height is infinitely often repeated and arbitrarily high bumps appear. For this last case, it would be sufficient to detect whether a bump is the highest one since the play is on the current stack level: indeed in a non strictly unbounded game, this happens infinitely often if and only if arbitrarily high bumps occur during the play. In order to detect this phenomena, we enrich the finite game graph of [23] with a counter that is incremented whenever a bump higher than the counter value is simulated, and that is decremented (mainly for

technical reasons) when a stair is simulated: if finitely many stair are simulated, the counter is incremented infinitely often if and only if arbitrarily high bumps occur on some fixed level (the one reached after the last stair). Therefore the unboundedness condition is simulated in this new one-counter game by requiring that either one simulates infinitely many stairs or the counter is infinitely often incremented.

Hence, when considering as winning condition a Boolean combination of a parity condition and of stack conditions, one gets a reduction to a one-counter game equipped with a simple combination of parity, Büchi and co-Büchi condition that can easily be expressed as a parity condition by slightly modifying the underlying one-counter process.

Before providing a description of the one-counter game graph $\widetilde{\mathcal{G}}$, let us consider the following informal description of this simulation game. We aim at simulating a play in the pushdown game from some initial vertex $(p_{in}, \perp)$. In $\widetilde{\mathcal{G}}$ we keep track of only the control state and the top stack symbol of the simulated configuration, and we maintain a counter κ. The interesting case is when it is in a control state p with top stack symbol α, and the player owning p wants to push a letter β onto the stack and change control state to q. For every strategy of Eve, there is a certain set of possible (finite) continuations of the play that will end with popping β from the stack. We require Eve to declare a vector $\overrightarrow{S} = ((S_0^-, S_0^+), \dots, (S_d^-, S_d^+))$ of $(d+1)$ pairs in $(2^Q)^2$, where S_i^- (*resp.* S_i^+) is the set of all states the game can be in after popping of β along these plays where in addition the stack height in the induced bump is strictly smaller (*resp.* equal or larger) than κ and the smallest visited color while β was on the stack is i.

Adam has two main choices. He can continue the game by pushing β onto the stack and update the state (we call this a *pursue* move). Otherwise, he can pick a set $S_i^\star$ (for $\star = -$ or $+$) and a state $s \in S_i^\star$, and continue the simulation from that state s (we call this a *jump* move). If he does a pursue move, then he remembers the vector $\overrightarrow{S}$ claimed by Eve and the counter κ is decreased; if later on a pop transition is simulated, the play stops and Eve wins if and only if the resulting state is in $S_\theta^\star$ where θ is the smallest color seen in the current level (this information is encoded in the control state, reset after each pursue move and updated after each jump move) and $\star = +$ if $\kappa = 0$ and $\star = -$ otherwise. If Adam does a jump move to a state s in $S_i^\star$, the currently stored value for θ is updated to $\min(\theta, i, col(s))$, which is the smallest color seen since the current stack level was reached, and if $\star = +$, the currently stored vector $\overrightarrow{R} = ((R_0^-, R_0^+), \dots, (R_d^-, R_d^+))$ is changed to $\overrightarrow{R^+} = ((R_0^+, R_0^+), \dots, (R_d^+, R_d^+))$ and κ is incremented.

Therefore the main vertices of the one-counter game graph are configurations of the form $[(p, \alpha, \overrightarrow{R}, \theta), \kappa]$ and they are controlled by the player that control p. Intermediate configurations are used to handle the previously described intermediate steps. The local structure is given in Figure 1 (circle vertex are those controlled by Eve). Two special control states *tt* and *ff* are used to simulate pop moves. This game graph is equipped with a coloring function on the vertices and on the edges: vertices of the form $[(p, \alpha, \overrightarrow{R}, \theta), \kappa]$ have color $col(p)$, edges leaving

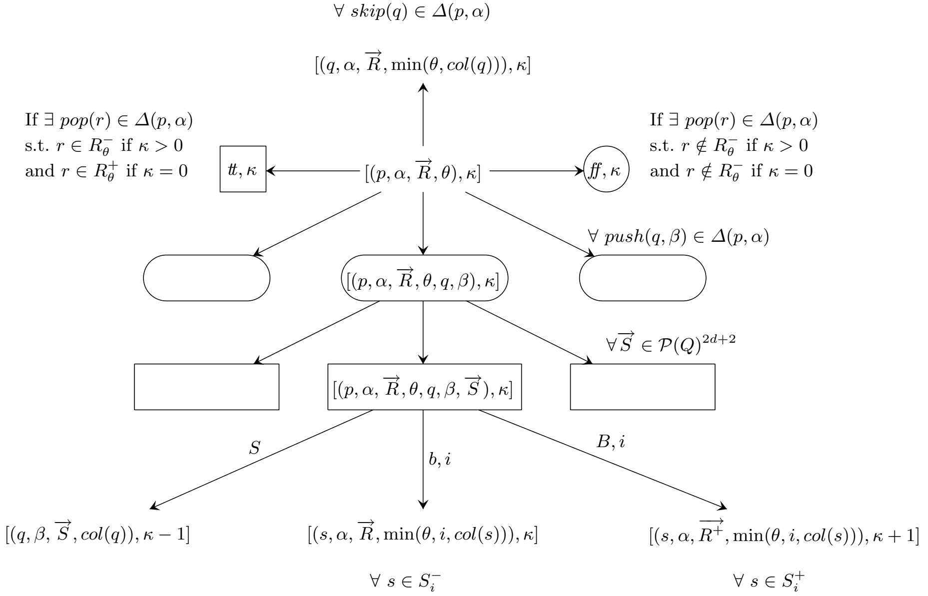


Fig. 1. Local structure of $\widetilde{\mathcal{G}}$

from a vertex $[(p, \alpha, \overrightarrow{R}, \theta, q, \beta, \overrightarrow{S}), \kappa]$ have two colors, one in $\{S, b, B\}$ (the color is S if the edge simulates a stair, b if it simulates a bump smaller than κ and B otherwise) and one in $\{0, \dots, d\}$ if it simulates a bump (the color is θ is the bump has color θ). It is easily seen that intermediate control states can be used to have only colors on vertices.

The winning condition for the game played on $\widetilde{\mathcal{G}}$ depends on the winning condition considered in the pushdown graph. If the winning condition is of the form $\psi(\Omega_1, \dots, \Omega_k)$ for a Boolean formula ψ, the winning condition on $\widetilde{\mathcal{G}}$ will be $\psi(\widetilde{\Omega_1}, \dots, \widetilde{\Omega_k})$, where

$$\widetilde{\Omega} = \begin{cases} \Omega_{par} & \text{if } \Omega = \Omega_{par} \\ \Omega_{Buc}(\{S, B\}) & \text{if } \Omega = \Omega_{Ubd} \\ \Omega_{co-Buc}(\{S, B\}) & \text{if } \Omega = \Omega_{Bd} \\ \Omega_{Buc}(\{S\}) & \text{if } \Omega = \Omega_{StUbd} \\ \Omega_{co-Buc}(\{S\}) & \text{if } \Omega = \Omega_{Rep} \end{cases}$$

Our main result is the following.

Theorem 5. *A configuration* $(p_{in}, \perp)$ *is winning for Eve in* $\mathbb{G} = (\mathcal{G}, \psi(\Omega_1, \dots, \Omega_k))$ *if and only if* $[(p_{in}, \perp, ((\emptyset, \emptyset), \dots, (\emptyset, \emptyset), col(p_{in})), 0]$ *is winning for Eve in* $\widetilde{\mathbb{G}} = (\widetilde{\mathcal{G}}, \psi(\widetilde{\Omega_1}, \dots, \widetilde{\Omega_k}))$*. Hence, deciding the winner in such a pushdown game can be done in* ExpSpace*.*

6 Conclusion

Refining the techniques from [11, 4], we have obtained a PSPACE algorithm to decide the winner in a one-counter parity game. As this problem was shown to be DP-hard, a remaining question is whether the complexity gap can be reduced.

As a corollary of our main result, we have improved the best known upper bound for the μ-calculus model-checking problem against one-counter processes.

We have shown how to use one-counter parity games to solve pushdown games equipped with winning conditions requiring both regular properties and stack height properties. We briefly mention here an extension of our result. In [14] pushdown games equipped with visibly pushdown winning conditions were considered. Such winning conditions capture all regular properties and several natural non-regular properties. In this setting, one can express the strict unboundedness condition but not the unboundedness one. The technique to solve these games is similar to the one for parity pushdown games: it uses a reduction to a parity game played on a finite game graph. One can easily show that the techniques of Section 5 can be adapted to solve pushdown games equipped with a winning condition combining a visibly pushdown condition with an unboundedness condition.

Acknowledgments. I would like to acknowledge the anonymous referees for their helpful suggestions and remarks.

References

1. A. Arnold and D. Niwiński. *Rudiments of mu-calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2001.
2. A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controlers with partial observation. *Theoretical Computer Science*, 303(1):7–34, 2003.
3. A. Bouquet, O. Serre, and I. Walukiewicz. Pushdown games with the unboundedness and regular conditions. In *Proceedings of FST&TCS'03*, volume 2914 of *LNCS*, pages 88–99. Springer, 2003.
4. T. Cachat. Two-way tree automata solving pushdown games. In E. Grädel, W. Thomas, and T. Wilke, editors, *Automata, Logics, and Infinite Games*, volume 2500 of *LNCS*, pages 303–317. Springer, 2002.
5. T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a Σ_3-winning condition. In *Proceedings of CSL'02*, volume 2471 of *LNCS*, pages 322–336. Springer, 2002.
6. E. A. Emerson, C. Jutla, and A. Sistla. On model-checking for fragments of mu-calculus. In *Proceedings of CAV'93*, volume 697 of *LNCS*, pages 385–396. Springer, 1993.
7. H. Gimbert. Parity and exploration games on infinite graphs. In Springer, editor, *Proceedings of CSL'04*, volume 3210 of *LNCS*, pages 56–70, 2004.
8. P. Jančar, A. Kučera, F. Moller, and Zdeněk Sawa. DP lower bounds for equivalence-checking and model-checking of one-counter automata. *Information and Computation*, 188:1–19, 2004.

9. A. Kucera. Efficient verification algorithms for one-counter processes. In Springer, editor, *Proceedings of ICALP'00*, volume 1853 of *LNCS*, pages 317–328, 2000.
10. O. Kupferman, N. Piterman, and M. Vardi. Extended temporal logic revisited. In Springer, editor, *Proceedings of Concur'01*, volume 2154 of *LNCS*, pages 519–535, 2001.
11. O. Kupferman and M. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proceedings of CAV'00*, volume 1855 of *LNCS*, pages 36–52. Springer, 2000.
12. O. Kupferman and M. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic*, 2(3):408–429, 2001.
13. C. Löding. Methods for the transformation of ω-automata: Complexity and connection to second order logic. Diplomata thesis, Christian-Albrechts-University of Kiel, 1998.
14. C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *Proceedings of FST&TCS'04*, volume 3328 of *LNCS*, pages 408–420. Springer, 2004.
15. D. A. Martin. Borel determinacy. *Annals of Mathematics*, 102(363–371), 1975.
16. C. Papadimitriou. *Complexity Theory*. Addison Wesley, 1994.
17. N. Piterman. Extending temporal logic with ω-automata. Master's thesis, The Weizmann Institute of Science, 2000.
18. O. Serre. *Contribution à l'étude des jeux sur des graphe de processus à pile*. PhD thesis, Université Paris VII, November 2004.
19. O. Serre. Games with winning conditions of high Borel complexity. In *Proceedings of ICALP'04*, volume 3142 of *LNCS*, pages 1150–1162. Springer, 2004.
20. O. Serre. Parity games played on transition graphs of one-counter processes: full version with complete proofs. `http://www.liafa.jussieu.fr/~serre`
21. W. Thomas. On the synthesis of strategies in infinite games. In *Proceedings of STACS 1995*, volume 900 of *LNCS*, pages 1–13. Springer, 1995.
22. M. Vardi. Reasoning about the past with two-way automata. In *Proceedings of ICALP 1998*, volume 1443 of *LNCS*, pages 628–641. Springer, 1998.
23. I. Walukiewicz. Pushdown processes: games and model checking. In *Proceedings of CAV'96*, volume 1102 of *LNCS*, pages 62–74. Springer, 1996.
24. I. Walukiewicz. Pushdown processes: games and model checking. *Information and Computation*, 157:234–263, 2000.
25. I. Walukiewicz. A landscape with games in the background. In *Proceeding of LICS'04*, pages 356–366. IEEE Computer Society, 2004.
26. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.

Bidomains and Full Abstraction for Countable Nondeterminism

James Laird*

Dept. of Informatics, University of Sussex, UK
jiml@sussex.ac.uk

Abstract. We describe a denotational semantics for a sequential functional language with random number generation over a countably infinite set (the natural numbers), and prove that it is fully abstract with respect to may-and-must testing.

Our model is based on biordered sets similar to Berry's bidomains, and stable, monotone functions. However, (as in prior models of unbounded non-determinism) these functions may not be continuous. Working in a biordered setting allows us to exploit the different properties of both extensional and stable orders to construct a Cartesian closed category of sequential, discontinuous functions, with least and greatest fixpoints having strong enough properties to prove computational adequacy.

We establish full abstraction of the semantics by showing that it contains a simple, first-order "universal type-object" within which all types may be embedded using functions defined by (countable) ordinal induction.

1 Introduction

Non-determinism is an abstract property with which we may represent the inherent uncertainty of a computational system, whether ocurring by accident or by design. When describing a non-deterministic system, we are typically interested in the possibility of failure, whether by divergence or premature termination. However, it is well known that capturing these behaviours in systems exhibiting unbounded non-determinism — i.e. programs which may choose between an infinite set of possible steps without diverging — presents a challenge for denotational semantics, because semantic functions characterizing their divergent behaviours are not, in general, continuous (see e.g. [1]). The object of this paper is to describe a domain-theoretic setting in which we may successfully capture the observable properties of functional programs with countable non-determinism via a semantics which is fully abstract with respect to may and must testing.

The basis for our model is a category of biordered sets and order-preserving functions based on Berry's *bidomains* [2]. In previous work by the author [8, 7] these have been used to give fully abstract models of sequential languages such as unary PCF (which may itself be considered as a λ-calculus with a binary choice

* Supported by UK EPSRC grant GR/S72181.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 352–366, 2006.
© Springer-Verlag Berlin Heidelberg 2006

operator). However, as well as capturing sequentiality, using two orders allows us to resolve some of the continuity problems associated with unbounded non-determinism. Essentially, they give separate extensional and intensional characterizations of programs, each having different completeness properties, which we exploit in proving (e.g.) computational adequacy.

Another feature of previous biorder-based semantics which is developed here is the focus on *observably sequentiality*, in which failure by divergence, and failure by premature termination (error) are distinguished. As observed by Cartwright and Felleisen [3], this simplifies the full abstraction problem for sequential functional languages, by making evaluation-order extensionally observable. In a non-deterministic setting, separating the two forms of failure makes a certain duality between "may" and "must" testing evident in both operational and denotational semantics: we give separate models of these in the same category of biorders, by interpreting error as a least element and recursion as a greatest fixed point with respect to may-testing, and interpreting error as a greatest element and recursion as a least fixed point with respect to must-testing.

We establish full abstraction for both may and must-testing semantics by developing a methodology used for proving definability results for observably sequential languages in particular (e.g. [11, 8]). We show that each type-object is a retract of a first-order "universal" type-object, and that these retractions are definable as terms in the language. This sheds some light on the process of computing interaction between functions with unbounded non-determinism, via countable sequences of unfoldings, in addition to sidestepping reliance of typical proofs of full abstraction on continuity and algebraicity.

1.1 Related Work

Apt and Plotkin [1] study a fully abstract denotational model of a simple imperative language with random assignment in a setting which brings together much of the preceding work on the semantics of countable non-determinism, and clarifies the role of non-continuity in particular. Lassen and Pitcher [9] study bisimulation equivalences based on may and must testing for a functional language similar to that modelled here. *Game semantics* has been used to describe denotational models of non-deterministic langages: Harmer and McCusker [6] have described a fully abstract may-and-must games model of Idealized Algol with bounded choice, whilst Levy [10] has described a game semantics of a language with unbounded non-determinism which captures an infinite trace equivalence. However, the biorder model described here appears to be the first fully abstract may-and-must testing semantics for a functional language with unbounded choice.

2 Syntax and Operational Semantics

We illustrate our approach by describing may and must semantics for a small functional language with countable non-determinism (which could be regarded as a target-language for CPS translation): a simply-typed λ-calculus with arithmetic, recursion and a random-number generator. Types are generated from two

ground types: a data type of natural number *values* and a program (or "response") type $\mathtt{o}$ containing no values, but a single "error" term. Programs of function type may take either data or programs as arguments, but must return a program — i.e. $\mathtt{nat}$ may not occur on the right of an arrow. Thus the types of our language are:

$$T ::= \mathtt{nat} \mid \mathtt{o} \mid T \Rightarrow P$$

where $P \neq \mathtt{nat}$ (we refer to non-$\mathtt{nat}$ types as *pointed*).

Terms are obtained by extending the simply-typed λ-calculus with a set of basic arithmetic constants and (primitive recursive) operations on $\mathtt{nat}$, including:

- zero ($\mathtt{0}$), successor and predecessor ($\mathtt{succ(_)}$, $\mathtt{pred(_)}$)
- equality testing, $_{=}_$
- "injective pairing" ($_ * _$) and projection $\mathtt{fst}_$ and $\mathtt{snd}_$.

and the following constants:

Error $\mathbf{e} : \mathtt{o}$,
Zero test $\mathtt{If0} : \mathtt{nat} \Rightarrow P \Rightarrow P \Rightarrow P$.
Fixpoints $\mathtt{Y} : (P \Rightarrow P) \Rightarrow P$.
Random number generation $\mathtt{rnd} : (\mathtt{nat} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o}$

We write $\mathtt{Eq}$ for $\lambda wxyz.((\mathtt{If0}\,(w{=}x))\,y)\,z : \mathtt{nat} \Rightarrow \mathtt{nat} \Rightarrow P \Rightarrow P \Rightarrow P$, and Ω for the divergent term $\mathtt{Y}\,\lambda x.x$ at each pointed type.

2.1 Operational Semantics

Note that any closed term $\mathtt{t} : \mathtt{nat}$ is an arithmetic expression derived from the total operations in the language. We assume an operation $|_|$ evaluating such expressions to numerals, which thus has the properties:

- $|\mathtt{s} = \mathtt{t}| = \mathtt{0}$ if $|\mathtt{s}| = |\mathtt{t}|$ and $|\mathtt{s} = \mathtt{t}| = \mathtt{1}$, otherwise.
- $|\mathtt{fst}(\mathtt{s} * \mathtt{t})| = |\mathtt{s}|$ and $|\mathtt{snd}(\mathtt{s} * \mathtt{t})| = |\mathtt{t}|$.

We define two evaluation relations $\Downarrow^{may}$ and $\Downarrow^{must}$ between closed terms of pointed type and "canonical forms" (λ-abstractions, $\mathtt{If0}$, $\mathtt{rnd}$ and $\mathbf{e}$) by combining the following, standard, "deterministic" fragment:

$$\frac{}{\mathbf{e} \Downarrow \mathbf{e}} \qquad \frac{}{\mathtt{rnd} \Downarrow \mathtt{rnd}} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}$$

$$\frac{}{\mathtt{If0} \Downarrow \mathtt{If0}} \qquad \frac{}{\mathtt{If0}\,\mathtt{t} \Downarrow \lambda xy.x}\ |\mathtt{t}| = \mathtt{0} \qquad \frac{}{\mathtt{If0}\,\mathtt{t} \Downarrow \lambda xy.y}\ |\mathtt{t}| \neq \mathtt{0}$$

$$\frac{}{\mathtt{Y} \Downarrow \lambda f.f\,(\mathtt{Y}\,f)} \qquad \frac{M \Downarrow \lambda x.M' \qquad M'[N/x] \Downarrow C}{M\,N \Downarrow C}$$

with one of the following rules for evaluating $\mathtt{rnd}$ by erratically generating a numeral and passing it to its argument:

$$\frac{\exists n {\in} \mathbb{N}.M\,\mathbf{n} \Downarrow^{may} \mathbf{e}}{\mathtt{rnd}\,M \Downarrow^{may} \mathbf{e}}\text{May} \qquad \frac{\forall n {\in} \mathbb{N}.M\,\mathbf{n} \Downarrow^{must} \mathbf{e}}{\mathtt{rnd}\,M \Downarrow^{must} \mathbf{e}}\text{Must}$$

We define notions of approximation and equivalence with respect to may and must testing. Given $M, N : T$:

- $M \lesssim^{may} N$ if for all compatible program contexts $C[\cdot] : \mathtt{o}$, $C[N]\Downarrow^{may}\mathbf{e}$ implies $C[M]\Downarrow^{may}\mathbf{e}$. $M \simeq^{may} N$ if $M \lesssim^{may} N$ and $N \lesssim^{may} M$.
- $M \lesssim^{must} N$ if for all compatible program contexts $C[\cdot] : \mathtt{o}$, $C[M]\Downarrow^{must}\mathbf{e}$ implies $C[N]\Downarrow^{must}\mathbf{e}$. $M \simeq^{must} N$ if $M \lesssim^{must} N$ and $N \lesssim^{must} M$.

Note the direction of the implication in the definition of may-approximation: $M \lesssim^{may} N$ if testing N leads to fewer errors than testing M.

As expected, there are functions which not continuous with respect to $\lesssim^{must}$ (considered as a partial order on $\simeq^{must}$ equivalence-classes of terms) — in particular, the operator $\mathtt{rnd}$ itself. For example, let $M_0 : \mathtt{nat} \Rightarrow \mathtt{o} = \lambda x.\Omega$ and $M_{i+1} = \lambda x.((\mathtt{If0}\, x)\, \mathbf{e})\, (M_i\, \mathtt{pred}(x))$ — i.e. M_i terminates if and only if its argument is less than i. So $\mathtt{rnd}\, M_i \not\Downarrow^{must}$ for all i, but the $\lesssim^{must}$ least upper bound of the chain $M_0 \lesssim^{must} M_1 \lesssim^{must} \ldots$ is $\lambda x.\mathbf{e}$, and $\mathtt{rnd}\, \lambda x.\mathbf{e}\Downarrow^{must}$. We study further examples of continuity and noncontinuity in Section 4.1.

The expressiveness of the language may be exploited by using it as the basis for CPS interpretation of more elaborate functional languages with unbounded nondeterminism. For example, we may translate PCF with random number generation simply by representing the type of natural number *computations*, $\mathtt{nat}_\perp^\top$, as $(\mathtt{nat} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o}$, giving $\mathtt{rnd} : \mathtt{nat}_\perp^\top$. (The corresponding bidomain model will contain additional elements corresponding to simple control operators and errors, yielding a fully abstract semantics of Cartwright and Felleisen's SPCF [3] with random number generation). Similarly, we may CPS translate Lassen and Pitcher's [9] version of Moggi's metalanguage extended with countable choice by representing the nondeterminism monad constructor $\mathsf{P}_$ as the continuations monad $(_ \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o}$. (Again, the translation and associated model will be fully abstract if first-class continuations are included in the source language.)

By distinguishing the two notions of failure, and taking them as the basis for our notions of observation in our model, we can also reason about a variety of behaviours of programs in such languages. For instance we may capture the requirement that $M : \mathtt{nat}_\perp^\top$ *may* converge to some (non-error) value as the conjunction of $M\, \lambda x.\mathbf{e}\Downarrow^{may}$ and $M\, \lambda x.\Omega \not\Downarrow^{must}$, and the requirement that M *must* converge to a value as $M\, \lambda x.\mathbf{e}\Downarrow^{must}$ and $M\, \lambda x.\Omega \not\Downarrow^{may}$.

3 Complete Meet Biorders

Berry's biorders [2, 4] are based on a binary greatest lower bound operator (i.e. a meet semi-lattice) which may be used to interpret binary choice [8]. Thus to give a semantics of unbounded choice, we develop a notion of biorder based on complete lattices (i.e. having a greatest lower bound operator for arbitrary sets).

Definition 1. *A complete (meet) biorder is a triple $\langle D, \sqsubseteq, \leq \rangle$ consisting of a set D with two partial orders $\sqsubseteq, \leq \subseteq D \times D$ such that:*

- $(D, \sqsubseteq)$ *(the extensional order) is a complete lattice: every subset X has a greatest lower bound $\bigsqcap X$.*
- $(D, \leq)$ *(the stable order) is included in $\sqsubseteq$ ($\leq \subseteq \sqsubseteq$), has a least element, $\bot = \bigsqcap D$ and for any $X, Y \subseteq D$ such that $X \leq Y$ in the Egli-Milner order (i.e. $\forall x \in X \exists y \in Y.x \leq y \wedge \forall y \in Y.\exists x \in X.x \leq Y$) we have $\bigsqcap X \leq \bigsqcap Y$.*

We shall write $\uparrow X$ if $X \subseteq D$ is non-empty[1] and bounded above in $(D, \leq)$, observing that $(D, \leq)$ is bounded co-complete in the following sense:

Lemma 1. *If $\uparrow X$ then $\bigsqcap X$ is a greatest lower bound for X in $(D, \leq)$.*

Proof. Suppose X is bounded above by y in $\leq$. Then for any $x \in X$, $X \leq \{x, y\}$ and so $\bigsqcap X \leq x \sqcap y = x$, and if z is a $\leq$-lower bound for X, then $\{z\} \leq X$ and so $z \leq \bigsqcap X$.

Products of complete meet biorders are defined by taking the pointwise orderings on the product of the underlying sets. Particular examples include the one-element biorder 1 (the unit for the product), the "Sierpinski" biorder Σ containing two elements, ordered stably and extensionally.

Definition 2. *A function f from $(D, \sqsubseteq, \leq)$ to $(D', \sqsubseteq', \leq')$ is monotone if it preserves both orders, and (completely) stable if for every stably bounded set X, $f(\bigsqcap X) = \bigsqcap f(X)$.*

Proposition 1. *The category of complete meet biorders and completely stable and monotone functions is Cartesian closed.*

Proof. For complete biorders $(D, \sqsubseteq_D, \leq_D)$ and $(E, \sqsubseteq_E, \leq_E)$ the function-space $D \Rightarrow E$ is the biorder over the set of monotone and stable functions from D to E in which the extensional order is defined:

$$f \sqsubseteq_{D \Rightarrow E} g \text{ if } f(x) \sqsubseteq_E g(x) \text{ for all } x \in D.$$

and the stable order is defined:

$$f \leq_{D \Rightarrow E} g \text{ if for all } x \leq_D y,\ f(x) \leq_E g(y) \text{ and } f(x) = f(y) \sqcap g(x).$$

This satisfies the axioms for a complete biorder, with the greatest lower bound of a bounded set of functions F defined pointwise: $(\bigsqcap F)(x) = \bigsqcap\{f(x) \mid f \in F\}$.

Thus we have the basis for the semantics of functional languages with unbounded choice (a CCC with a greatest lower bound operator). To interpret the `Y` combinator we require least and greatest fixed points of each endomorphism $f : D \to D$. As in [1], we may compute these as the suprema/infima of chains of approximants obtained by iterating f countably many times.

Proposition 2. *Every endomorphism $f : A \to A$ has a $\sqsubseteq$-least fixed point $f^\dagger : \mathbf{1} \to A$ and a $\sqsubseteq$-greatest fixed point $f^\ddagger : \mathbf{1} \to A$.*

[1] In particular, $\top = \bigsqcup \varnothing$ is not in general a $\leq$-greatest element.

Proof. We obtain $f^\dagger$ as a stationary point of the $\sqsubseteq$-chain defined $f^\lambda = f(\bigsqcup_{\kappa<\lambda} f^\kappa)$ for each ordinal λ. Then $\lambda \leq \kappa$ implies $f^\lambda \sqsubseteq f^\kappa$, and if $f^\lambda < f^{\lambda+1}$ then $f^\kappa \sqsubset f^\mu$ for all $\kappa < \mu \leq \lambda$. So if κ has cardinality strictly greater than A, then we must have $f(f^\kappa) = f^\kappa$. Moreover f^κ is a least (pre)fixed point, since if $f(a) \sqsubseteq a$ then $f^\lambda \sqsubseteq a$ for all λ.

We construct the greatest fixed point $f^\ddagger$ similarly, as a stationary point in the descending $\sqsubseteq$-chain defined $f^\lambda = f(\bigsqcap_{\kappa<\lambda} f^\kappa)$.

However, since least upper bounds in complete meet biorders are defined indirectly, the mere existence of the least fixed point $f^\dagger$ is not sufficient to prove that it yields an interpretation of `Y` which is *computationally adequate*. It transpires that the continuity property required to prove adequacy is that for $f : (A \Rightarrow B) \rightarrow (A \Rightarrow B)$, $(\bigsqcup_{\kappa<\lambda} f^\kappa)(e) = \bigsqcup_{\kappa<\lambda} f(e)$. In general, the least upper bound of a $\sqsubseteq$-directed set of functions *cannot* be determined in this way (i.e. it is not the case that $(\bigsqcup F)(x) = \bigsqcup\{f(x) \mid f \in F\}$)— we give an example in the next section. However, we shall now show that we may define a full (Cartesian closed) subcategory of biorders in which the stable order is a cpo in which least upper bounds of directed sets of functions is determined pointwise.

Definition 3. *A complete meet biorder D is a complete meet bidomain*[2] *if it satisfies the following conditions:*

Stable Completeness. *Every set $X \subseteq D$ which is* stably directed *(i.e. upwards directed with respect to $\leq$) has a least upper bound $\bigvee X$ with respect to the stable order, such that $\bigvee X = \bigsqcup X$, and satisfying the following distributivity property:*
for any y with $y \uparrow \bigvee X$, $y \sqcap \bigvee X = \bigvee\{x \sqcap y \mid x \in X\}$.

Algebraicity. *An element $c \in D$ is* weakly compact *($c \in \mathcal{K}(D)$) if for every stably directed set X such that $c \sqsubseteq \bigvee X$ there exists $x \in X$ such that $c \sqsubseteq x$. D is (weakly) algebraic if every element in $d \in D$ is the ($\sqsubseteq$) least upper bound of its set of weakly compact approximants — $d = \bigsqcup\{c \in \mathcal{K}(D) \mid c \sqsubseteq d\}$.*

Lemma 2. *If D, E are stably complete and algebraic, then $D \Rightarrow E$ is stably complete.*

Proof. Given a stably directed set of functions F, the set $\{f(x) \mid f \in F\}$ is stably directed, so we may define the stable supremum of F pointwise: $(\bigvee F)(x) = \bigvee\{f(x) \mid f \in F\}$.

This is monotone — if $x \leq y$ then for all f, $f(x) \leq f(y) \leq (\bigvee F)(y)$ and so $(\bigvee F)(x) \leq (\bigvee F)(y)$ — and binary-stable: if $x \uparrow y$, then $(\bigvee F)(x) \sqcap (\bigvee F)(y) = \bigvee\{f(x) \sqcap \bigvee F(y) \mid f \in F\} = \bigvee\{f(x) \sqcap g(y) \mid f, g \in F\} \sqsubseteq \bigvee F(x \sqcap y)$.

To show that $\bigvee F$ is stable with respect to infima of stably bounded infinite sets, it is sufficient to show that it preserves infima of (downwards) stably directed sets. So suppose we have a downwards stably-directed set X. We need to show that $\bigsqcap(\bigvee F)(X) \sqsubseteq \bigvee F(\bigsqcap X)$.

[2] Note that a complete meet bidomain need not be a bidomain in the sense of Berry.

Suppose c is a compact element such that $c \sqsubseteq \bigsqcap(\bigvee F)(X)$. Choosing $x \in X$, we have $c \sqsubseteq (\bigvee F)(x)$ and so by compactness of c, there exists $f \in F$ such that $c \sqsubseteq f(x)$. Then for any $y \in X$, there exists $z \in X$ such that $z \leq x, y$, and so we may find $g \in F$ such that $c \sqsubseteq g(z)$, and $h \in F$ such that $f, g \leq h$. Thus $f(z) = f(x) \sqcap h(z)$, and so $c \sqsubseteq f(z) \sqsubseteq f(y)$. So $c \sqsubseteq \bigsqcap f(X)$, and $c \sqsubseteq \bigvee F(\bigsqcap X)$ as required.

Moreover $\bigvee F$ is a $\leq$-least upper bound for F (as well as being a $\sqsubseteq$ least upper bound): if $f \in F$ then for all x, $f(x) \leq (\bigvee F)(x)$ and if $x \leq y$ then $f(y) \sqcap (\bigvee F)(x) = \bigvee\{f(y) \sqcap g(x) \mid g \in F \sqcap f \leq g\} = f(x)$. If $f \leq g$ for all $f \in F$, then for all x, $(\bigvee F)(x) \leq g(x)$, and if $x \leq y$ then $(\bigvee F)(y) \sqcap g(x) = \bigvee\{f(y) \sqcap g(x) \mid f \in F\} = (\bigvee F)(x)$.

Since $\sqcap, \bigvee$ are both determined pointwise, the distributivity condition is straightforward to verify.

Lemma 3. *The complete meet bidomains and completely stable and monotone functions form a CCC.*

Proof. By Lemma 2, if D, E are complete meet bidomains then $D \Rightarrow E$ is stably complete, so it remains to show weak algebraicity. Given $f \in D \Rightarrow E$, $d \in D$, and weakly compact $c \in E$ such that $c \sqsubseteq f(d)$, we define $f_c^d \in D \Rightarrow E$ such that $f_c^d(x) = c$ if $d \sqsubseteq x$ and $f_c^d(x) = \bot$ otherwise.

Then f_d^c is monotone and completely stable: if $x \sqsubseteq y$, then if $d \sqsubseteq x$, $f_c^d(x) = f_c^d(y) = c$, otherwise $f_c^d(x) = \bot \leq f_c^d(y)$. Given a stably bounded set X, if $d \sqsubseteq \bigsqcap X$ then $d \sqsubseteq x$ for all $x \in X$, and so $f_c^d(\bigsqcap X) = c = \bigsqcap f_c^d(X)$. If $d \not sqleq \bigsqcap X$ then there exists $x \in X$ such that $d \not\sqsubseteq x$, and so $f_c^d(x) = \bot = f_c^d(\bigsqcap X)$.

It is straightforward to check that f_d^c is weakly compact (if $f_c^d \sqsubseteq \bigvee F$ then $f_c^d(d) = c \sqsubseteq (\bigvee F)(d)$ and so $c \sqsubseteq f(d)$ for some $f \in F$, and so $f_c^d \sqsubseteq f$) and $f = \bigsqcup\{f_c^d \mid d \in D \wedge c \in \mathcal{K}(E) \wedge c \sqsubseteq f(d)\}$.

To interpret unpointed types (in the current setting, just the type `nat` of natural number values), we define a notion of "pre-bidomain".

Definition 4. *A (complete meet) pre-bidomain* $(D, \sqsubseteq, \leq)$ *is a set* D *with partial orders* $\leq \subseteq \sqsubseteq$ *such that for each* $x \in D$, $D_x = \{y \in D \mid \exists z \in D. z \sqsubseteq x, y\}$ *is a co-complete bidomain.*

The co-product of pre-bidomains (formed pointwise) is a pre-bidomain, and gives the following characterization result.

Lemma 4. *For a pre-bidomain* D, *let* $\lfloor D \rfloor$ *be the set of* $\sqsubseteq$*-minimal elements of* D. *Then* $D \cong \coprod_{x \in \lfloor D \rfloor} D_x$.

Proof. Let $\bot(x) = \bigsqcap\{y \in D \mid y \sqsubseteq x\}$. Then for each x, $\bot(x)$ is a minimal element of D, and it is straightforward to show that the map sending x to $\mathsf{in}_{\bot(x)}(x)$ is an order-isomorphism.

Proposition 3. *The category of pre-bidomains and monotone and stable functions is bicartesian closed.*

Proof. We define the cartesian closed structure as for the category of complete bidomains and monotone and stable functions: thus the principal point to check is that the function-space yields a well-defined pre-bidomain, for which we use the decomposition into co-products (Lemma 4). We show that:

- for any complete meet bidomain A, and pre-bidomain D, $\coprod_{x \in \lfloor D \rfloor}(A \Rightarrow D_x) \cong A \Rightarrow \coprod_{x \in \lfloor D \rfloor} D_x \cong A \Rightarrow D$, and hence $A \Rightarrow D$ is a pre-bidomain.
- for any pre-bidomains D, E: $\Pi_{x \in \lfloor D \rfloor}(D_x \Rightarrow E) \cong (\coprod_{x \in \lfloor D \rfloor} D_x) \Rightarrow E \cong D \Rightarrow E$, and so $D \Rightarrow E$ is a pre-bidomain. (So if E is a complete bidomain then so is $D \Rightarrow E$.)

4 Denotational Semantics

We now give the may and must testing semantics of the functional language defined in Section 2. We interpret `nat` as the pre-bidomain $\mathbb{N}_* = \coprod_{i \in \mathbb{N}} 1$, and the remaining (pointed) types as the corresponding bidomains: i.e. $[\![o]\!] = \Sigma$ and $[\![S \Rightarrow T]\!] = [\![S]\!] \Rightarrow [\![T]\!]$.

We interpret terms-in-context $x_1 : S_1, \ldots, x_n : S_n \vdash M : T$ as monotone and completely stable functions from $[\![S_1]\!] \times \ldots [\![S_n]\!]$ to $[\![T]\!]$, giving two denotations $[\![M]\!]_{may}, [\![M]\!]_{must}$ for each term. We use the Cartesian closed structure to interpret λ-abstraction and application in standard fashion, and the associated operations on $\mathbb{N}$ to interpret the arithmetic constants and operations. Random assignment `rnd` is interpreted as the function which takes every argument except $\top$ to $\bot$:

$$[\![\texttt{rnd}]\!]_{may}(f) = [\![\texttt{rnd}]\!]_{must}(f) = \bigsqcap\{f(n) \mid n \in \mathbb{N}_*\}$$

Thus every program with neither recursion nor explicit errors has the same denotation in the may and must semantics.

In the may-testing semantics, we interpret the error as the *least* element $\bot$, and the fixpoint combinator $\mathtt{Y} : (P \Rightarrow P) \Rightarrow P$ as the *greatest* fixed point of the endomorphism $F : (P \Rightarrow P) \Rightarrow P \rightarrow (P \Rightarrow P) \Rightarrow P$ sending f to $\lambda g.g(f\ g)$. In the must-testing semantics we interpret the error as the *greatest* element $\top$, and $\mathtt{Y} : (P \Rightarrow P) \Rightarrow P$ as the *least* fixed point of F.

4.1 Examples

We give some examples of the continuity and noncontinuity properties of our model.

Noncontinuity. We have shown that the random number generator `rnd` is not continuous wiith respect to must-approximation. The same example suffices to show that its denotation (which we shall write as rnd) is not continuous with respect to extensional order nor the stable order. If we define $f_i : \mathbb{N} \Rightarrow \Sigma$ by $f_i(n) = \top$ if $n < i$ (so $[\![M_i]\!] = f_i$) then $f_i \leq f_{i+1}$ for all i. $\mathsf{rnd}(f_i) = \bot$ for all $i \in \omega$, but $\mathsf{rnd}(\bigvee\{f_i \mid i \in \omega\}) = \mathsf{rnd}(\top) = \top$.

Stable continuity of function application. We now give an example of a least upper bound of a stably-directed set of functions, defined pointwise. Let $g_i : ((\mathbb{N} \Rightarrow \Sigma) \Rightarrow \Sigma) \Rightarrow \Sigma$ be defined: $g_i(h) = h(f_i)$. Then $g_i \leq g_{i+1}$ for all $i \in \omega$ (since $f_i \leq f_{i+1}$), and so we may define the least upper bound $G = \bigvee\{g_i \mid i \in \omega\}$: $G(h) = \top$ if there exists i such that $h(f_i) = \top$. Note that G is distinct from the function $G'(h) = h(\bigsqcup\{f_i \mid i \in \omega\}) = h(\top)$, since $G(\mathsf{rnd}) = \bot$ and $G'(\mathsf{rnd}) = \top$.

Moreover, G is definable in our language — it is the denotation of $\lambda h.(\mathtt{Y}\, \lambda f.\lambda x.h\, (\lambda y.\mathtt{If0}\, (y < x)\ \text{ then } \top \text{ else } (f\, y)))\, 0$.

Extensional noncontinuity of function application. By contrast, we may observe that the least upper bound of a $\sqsubseteq$-chain of functions may not be determined pointwise. Define $h_i : (\mathbb{N} \Rightarrow \Sigma) \Rightarrow \Sigma$ by $h_i(f) = \prod_{n\in\omega} f(n+i)$ (i.e. h_i is the denotation of the term $\lambda f.\mathtt{rnd}\, \lambda x.f\ (x + \mathtt{n})$. Then $h_i \sqsubseteq h_{i+1}$ for each i (but $h_i \not\leq h_{i+1}$). The least upper bound of $\{h_i \mid i \in \omega\}$ is $\top$. (To show this, define $k_i : \mathbb{N} \Rightarrow \Sigma$ by $k_i(n) = \bot$ if $i < n$ and $k_i(n) = \top$, otherwise. Then $h_i(k_i) = \top$, and so if H is an upper bound for $\{h_i \mid i \in \omega\}$, $H(k_i) = \top$ for all i. So by stability, $H(\bot) = H(\prod\{k_i \mid i \in \omega\}) = \prod\{H(k_i) \mid i \in \omega\} = \top$.) So $(\bigsqcup\{h_i \mid i \in \omega\})(\bot) = \top$, but $\bigsqcup\{h_i(\bot) \mid i \in \omega\} = \bot$.

4.2 Inequational Soundness

Proposition 4. *$M{\Downarrow}^{may}C$ implies $[\![M]\!]_{may} = [\![C]\!]$ and $M{\Downarrow}^{must}C$ implies $[\![M]\!]_{must} = [\![C]\!]$.*

Proof. Both cases are proved by induction on the derivation of $M \Downarrow C$: in the case of must-testing we decorate the judgement $\Downarrow$ with an ordinal (upper) bound on the depth of its derivation, following the schema:

$$\frac{}{M{\Downarrow}^{\lambda}C} \qquad \frac{M{\Downarrow}^{\lambda}\lambda x.M' \qquad M'[N/x]{\Downarrow}^{\kappa}C}{M\,N{\Downarrow}^{\kappa}C}\ \kappa < \lambda \qquad \frac{\forall n{\in}\mathbb{N}.M\ \mathtt{n}{\Downarrow}^{\kappa}\mathbf{e}}{\mathtt{rnd}\,M{\Downarrow}^{\lambda}\mathbf{e}}\ \kappa < \lambda$$

Then if $M \Downarrow C$, $M \Downarrow^{\lambda} C$ for some λ, and we may prove by ordinal induction that if $M \Downarrow^{\lambda} C$ then $[\![M]\!]_{must} = [\![C]\!]$.

Proposition 5 (Adequacy). *$[\![M]\!]_{may} = \bot$ implies $M{\Downarrow}^{may}\mathbf{e}$ and $[\![M]\!]_{must} = \top$ implies $M{\Downarrow}^{must}\mathbf{e}$.*

Proof. The proofs for both models are essentially the same: we sketch the case for must-testing. This uses "approximation relations" in the style of Plotkin [12]: first we define a relation $\lhd_T$ between elements of $[\![T]\!]$ and closed terms of type T for each T:

- $n \lhd_{\mathtt{nat}} M$ if $|M| = \mathtt{n}$.
- $e \lhd_{\mathsf{o}} M$ if $e = \top$ implies $M{\Downarrow}^{must}\mathbf{e}$.
- $f \lhd_{S\Rightarrow T} M$ if $e \lhd_S N$ implies $f(e) \lhd_T M\,N$.

We then define $f : [\![\Gamma]\!] \to [\![T]\!] \lhd_{\Gamma,T} \Gamma \vdash M : T$ if $\Gamma = x_1 : S_1, \ldots, x_n : S_n$ and for all $e_1 \lhd_{S_1} N_1, \ldots, e_n \lhd_{S_n} N_n$ implies $f(e_1, \ldots, e_n) \lhd_T M[N_1/x_1, \ldots, N_n/x_n]$.

We prove that if $\Gamma \vdash M : T$ then $[\![M]\!]_s \lhd_{\Gamma,T} M$ by a standard structural induction. The only potentially problematic case is the fixpoint combinator $\mathtt{Y}$, for which we use the following observations:

For any (closed) $M : T$, the set $\{e \in T \mid e \lhd_T M\}$ is (stably) chain complete, since the least upper bound of a stable chain of functions is determined pointwise. Note also that $e \lhd_P M\,(\mathtt{Y}\,M)$ implies $e \lhd_P \mathtt{Y}\,M$.

To prove $[\![\mathtt{Y}]\!] \lhd_{(P \Rightarrow P) \Rightarrow P} \mathtt{Y}$, we show that $F^\lambda \lhd_{(P \Rightarrow P) \Rightarrow P} \mathtt{Y}$ for all λ by induction on λ. For the induction case, assume $F^\kappa \lhd_{(P \Rightarrow P) \Rightarrow P} \mathtt{Y}$ for all $\kappa < \lambda$, and hence $\bigvee_{\kappa<\lambda} F^\kappa \lhd_{(P \Rightarrow P) \Rightarrow P} \mathtt{Y}$ by stable chain completeness. Suppose $f \lhd_{P \Rightarrow P} M$. Then $F^\lambda(f) = f((\bigvee_{\kappa<\lambda} F^\kappa)(f)) \lhd_P M\,(\mathtt{Y}\,M)$, and so $F^\lambda(f) \lhd_P \mathtt{Y}\,M$ as required.

Corollary 1 (Inequational Soundness). *$[\![M]\!]_{may} \sqsubseteq [\![N]\!]_{may}$ implies $M \lesssim^{may} N$. $[\![M]\!]_{must} \sqsubseteq [\![N]\!]_{must}$ implies $M \lesssim^{must} N$.*

Proof. Suppose e.g. $[\![M]\!]_{must} \sqsubseteq [\![N]\!]_{must}$. Then for any compatible context $C[_]$, $C[M] \Downarrow$ implies $[\![C[M]]\!]_{must} = \top$ implies $[\![C[N]]\!]_{must} = \top$ implies $C[N] \Downarrow$ as required.

5 Full Abstraction

It remains to prove (inequational) completeness: we shall say that completeness holds at type T if for all closed $M, N : T$, if $M \lesssim^{may} N$ then $[\![M]\!]_{may} \sqsubseteq [\![N]\!]_{may}$ and if $M \lesssim^{must} N$ then $[\![M]\!]_{must} \sqsubseteq [\![N]\!]_{must}$.

So, for instance, completeness holds at $\mathtt{nat}$, since e.g. if $M \lesssim^{may} N$ then $(((\mathtt{Eq}\,M)\,\mathtt{n})\,\mathbf{e})\,\Omega{\Downarrow}^{may}$ implies $(((\mathtt{Eq}\,N)\,\mathtt{n})\,\mathbf{e})\,\Omega{\Downarrow}^{may}$, and hence $[\![M]\!]_{may} = [\![N]\!]_{may}$.

Lemma 5. *Completeness holds at the type* $\mathtt{nat} \Rightarrow \mathtt{o} \Rightarrow \mathtt{o}$.

Proof. Suppose e.g. $M \lesssim^{must} N$. Then by soundness and adequacy, for any $d \in \mathbb{N}$ and $e \in \{\top, \bot\}$ we have $([\![M]\!]_{must}\,d)\,e = \top$ implies $([\![N]\!]_{must}\,d)\,e = \top$, and so $[\![M]\!]_{must} \sqsubseteq [\![N]\!]_{must}$.

We reduce completeness at all pointed types to completeness at $\mathtt{nat} \Rightarrow \mathtt{o} \Rightarrow \mathtt{o}$ using the notion of *definable retraction*.

Definition 5. *Given types S, T, we write $[\![S]\!] \unlhd [\![T]\!]$ (with respect to an interpretation $\mathcal{M}$) if there is a retraction from $[\![S]\!]$ to $[\![T]\!]$ definable in $\mathcal{M}$: i.e. a pair of (closed) terms $(\mathtt{in} : S \Rightarrow T, \mathtt{out} : T \Rightarrow S)$ such that $[\![x : S \vdash \mathtt{out}\ (\mathtt{in}\ x) : S]\!]_{\mathcal{M}} = \mathtt{id}_{[\![S]\!]}$.*

Henceforth, unless noted otherwise, we will take $[\![S]\!] \unlhd [\![T]\!]$ to mean that there is a retraction definable in both may and must interpretations.

For example, we have $\mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow [\![P]\!] \unlhd \mathbb{N}_* \Rightarrow [\![P]\!]$ for any $[\![P]\!]$ via the definable retraction $(\lambda f.\lambda x.((f\ \mathtt{fst}(x))\ \mathtt{snd}(x)), \lambda g.\lambda y.\lambda z.g\ (y * z))$. Note that if $(\mathtt{in}_1, \mathtt{out}_1)$ and $(\mathtt{in}_2, \mathtt{out}_2)$ are definable retractions from $[\![S_1]\!]$ to $[\![S_2]\!]$ and from $[\![T_1]\!]$ to $[\![T_2]\!]$, then $(\lambda f.\lambda x.\mathtt{in}_2(f\ (\mathtt{out}_1\ x)), \lambda f.\lambda x.\mathtt{out}_2(f\ (\mathtt{in}_1\ x)))$ is a definable retraction from $[\![S_1 \Rightarrow T_1]\!]$ to $[\![S_2 \Rightarrow T_2]\!]$.

Let $U = \mathbb{N}_* \Rightarrow \Sigma \Rightarrow \Sigma$ — i.e. $U = [\![\mathtt{nat} \Rightarrow \mathtt{o} \Rightarrow \mathtt{o}]\!]$. We will show that U is *universal* amongst the pointed type-objects — i.e. $[\![P]\!] \trianglelefteq U$ for all pointed types P. This is sufficient to prove completeness at all types.

Lemma 6. *If $[\![S]\!] \trianglelefteq U$ in $\mathcal{M}$ then $\mathcal{M}$ is complete at type S.*

Proof. If $M \lesssim^{\mathcal{M}} N$ then $\mathtt{in}\, M \lesssim^{\mathcal{M}} \mathtt{in}\, N$ and so $[\![\mathtt{in}\, M]\!]_{\mathcal{M}} \sqsubseteq [\![\mathtt{in}\, N]\!]_{\mathcal{M}}$ and so $[\![M]\!]_{\mathcal{M}} = \mathsf{out}[\![\mathtt{in}\, M]\!]_{\mathcal{M}} \sqsubseteq \mathsf{out}[\![\mathtt{in}\, N]\!]_{\mathcal{M}} = [\![N]\!]_{\mathcal{M}}$ as required.

We will use the fact that we may regard elements of $\mathbb{N}_* \Rightarrow \Sigma$ as infinite lists of elements of Σ: for $M : \mathtt{o}, N : \mathtt{nat} \Rightarrow \mathtt{o}$, we define $M :: N : \mathtt{nat} \Rightarrow \mathtt{o} = \lambda x.((\mathtt{If0}\, x)\, M)\, (N\, \mathtt{pred}(x))$, $\mathtt{hd} : (\mathtt{nat} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o} = \lambda f.f\, \mathtt{0}$ and $\mathtt{tl} : (\mathtt{nat} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{nat} \Rightarrow \mathtt{o} = \lambda f.\lambda x.f\, \mathtt{succ}(x)$. Then $[\![\mathtt{hd}\, (M :: N) = M]\!]$ and $[\![\mathtt{tl}\, (M :: N)]\!] = [\![N]\!]$.

Lemma 7. $\Sigma \Rightarrow (\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma \trianglelefteq (\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma$.

Proof. The retraction is definable via the terms $\mathtt{in} = \lambda f.\lambda g.(f\, (\mathtt{hd}\, g))\, (\mathtt{tl}\, g)$ and $\mathtt{out} = \lambda h.\lambda x.\lambda k.h\, (x :: k)$.

Definition 6. *Given $e \in \mathbb{N}_* \Rightarrow A$, $n \in \mathbb{N}_*$ and $d \in A$ let $e[d]_n \in \mathbb{N}_* \Rightarrow A$ (the "n-insertion" of d into e) be defined:*

- *$e[d]_n(m) = d$ if $n = m$,*
- *$e[d]_n(m) = e(m)$, otherwise.*

For terms $M : \mathtt{nat} \Rightarrow T$, $N : T$ and $t : \mathtt{nat}$, we define the coresponding term $M[N]_t : \mathtt{nat} \Rightarrow T = \lambda x.(((\mathtt{Eq}\, t)\, x)\, (N))\, (M\, x)$. We use insertion to define another key retraction.

Lemma 8. $(\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma \trianglelefteq \mathbb{N}_* \Rightarrow \Sigma \Rightarrow \Sigma$.

Proof. (Must testing case). Let $\mathsf{in} = \lambda f.\lambda x.\lambda y.f\, \lambda z.(((\mathtt{Eq}\, x)\, z)\, y)\, \mathbf{e}$ and $\mathtt{out} = \lambda f.\lambda g.\mathtt{rnd}\, \lambda x.(f\, x)\, (g\, x)$.

For any $g : \mathbb{N}_* \to \Sigma$, the set $\{\top[g(n)]_n \mid n \in \mathbb{N}_*\}$ is stably bounded above by the constantly $\top$ function, and $g = \bigsqcap\{\top[g(n)]_n \mid n \in \mathbb{N}_*\}$. Thus $(\mathtt{out}\, \mathsf{in}(f))(g) = \bigsqcap\{f(\top[g(n)]_n) \mid n \in \mathbb{N}_*\} = f(\bigsqcap\{\top[g(n)]_n \mid n \in \mathbb{N}_*\} = f(g)$ by stability.

We will now show that $U \Rightarrow \Sigma \trianglelefteq \mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow (\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma$, and hence by Lemma 8, $U \Rightarrow \Sigma \trianglelefteq U$. The key to defining this retraction is the *sequentiality* of the function-space $U \Rightarrow \Sigma$.

Definition 7. *Given $f \in (\mathbb{N}_* \Rightarrow A) \Rightarrow \Sigma$, where A is a complete bidomain, we say that f is i-strict if for all $g \in \mathbb{N}_* \Rightarrow A$, $g(i) = \bot$ implies $f(g) = \bot$. We write $\mathsf{strict}(f)$ for the set of $i \in \mathbb{N}$ such that f is i-strict.*

Lemma 9 (Sequentiality). *For any complete bidomain A, every $f \in (\mathbb{N}_* \Rightarrow A) \Rightarrow \Sigma$ is constant or i-strict for some i.*

Proof. Note that the set $\{\top[\bot]_i \mid i \in \mathbb{N}\} \subseteq \mathbb{N}_* \Rightarrow A$ is stably bounded above by $\top$. Suppose $f \neq \top$. Then $f(\bot) = f(\bigsqcap\{\top[\bot]_i \mid i \in \mathbb{N}_*\}) = \bot$. So by stability $\bigsqcap\{f(\top[\bot]_i) \mid i \in \mathbb{N}_*\} = \bot$. Hence $f(\top[\bot]_i) = \bot$ for some i, and $g(i) = \bot$ implies $g \sqsubseteq \top[\bot]_i$ and so $f(g) = \bot$ — i.e. f is i-strict as required.

Let $I \in \Sigma \Rightarrow \Sigma$ be the identity function (note that $I \sqsubseteq \top$, but $I \not\leq \top$).

Definition 8. *Given* $f \in U \Rightarrow \Sigma$ *and* $n \in \mathbb{N}_*$, *let* $f_n = \lambda x.((x\,n)\,(f\,x[I]_n)) \sqcap (f\,x[\top]_n)$.

Lemma 10. *If* f *is* n*-strict then* $f = f_n$.

Proof. Consider $e \in \mathbb{N}_* \Rightarrow \Sigma \Rightarrow \Sigma$.

If $e(n) = \bot$, then by n-strictness of f, $f(e) = \bot$, and $f_n(e) = (\bot\, f(e[I]_n) \sqcap f(e[\top]_n) = \bot$.

If $e(n) = I$, then $e = e[I]_n$, and $f_n(e) = (I\, f(e[I]_n) \sqcap f(e[\top]_n) = f(e) \sqcap f(e[\top]_n) = f(e)$ by monotonicity of f.

If $e(n) = \top$, then $e = e[\top]_n$, and $f_n(e) = (\top\, f(e[I]_n) \sqcap f(e[\top]_n) = \top \sqcap f(e) = f(e)$.

Thus we can represent any (non-constant) $f \in U \Rightarrow \Sigma$ as a strictness index n, together with the two functions $\lambda x.f\,x[I]_n$ and $\lambda x.f\,x[\top]_n$ which (as we shall show) may be computed using a strictly smaller part of their argument.

Lemma 11. *Suppose* $e(n)(\top) = e(n)(\bot)$ *for all* $n \in \mathbb{N}_*$. *Then for any* $f \in U \Rightarrow \Sigma$, $f(e) = \bigsqcap\{e(n)(\bot) \mid n \in \mathsf{strict}(f)\}$.

Proof. If $e(n)(\bot) = \bot$ for some $n \in \mathsf{strict}(f)$, then $f(e) = f_n(e) = (e(n)(f(e[I]_n))) \sqcap f(e[\top]_n) = \bot \sqcap f(e[\top]_n) = \bot$. So suppose $e(n)(\bot) = \top$ for all $n \in \mathsf{strict}(f)$. Then $e \sqsupseteq \bigsqcap\{\top[\bot]_n \mid n \notin \mathsf{strict}(f)\}$ and so $f(e) \sqsupseteq f(\bigsqcap\{\top[\bot]_n \mid n \notin \mathsf{strict}(f)\}) = \bigsqcap\{f(\top[\bot]_n) \mid n \notin \mathsf{strict}(f)\} = \top$ by stability.

We also require an "injective pairing" operation on $\mathbb{N}_* \Rightarrow A \Rightarrow \Sigma$, derived from the fact that that $\mathbb{N}_* \Rightarrow A \Rightarrow \Sigma \cong (A \Rightarrow \Sigma)^\omega \cong (A \Rightarrow \Sigma)^\omega \times (A \Rightarrow \Sigma)^\omega$.

Definition 9. *Given* $M, N : \mathtt{nat} \Rightarrow T \Rightarrow \mathtt{o}$, *let* $\langle M, N\rangle : \mathtt{nat} \Rightarrow T \Rightarrow \mathtt{o} = \lambda x.\lambda y.((\mathtt{If0}\,\mathtt{fst}(x))\,((M\,\mathtt{snd}(x))\,y))\,((N\,\mathtt{snd}(x))\,y)$ *and* $\pi_i : (\mathtt{nat} \Rightarrow T \Rightarrow \mathtt{o}) \Rightarrow \mathtt{nat} \Rightarrow T \Rightarrow \mathtt{o} = \lambda f.\lambda x.\lambda y.(f\ (\mathtt{i} * x))\,y$. *Then* $\pi_i\,\langle M_0, M_1\rangle = M_i$ *for* $i \in \{0, 1\}$.

We finally note that $\sqcap$ is definable as erratic binary choice: given $M, N : \mathtt{o}$: $M\,\mathtt{or}\,N : \mathtt{o} = \mathtt{rnd}\,\lambda x.((\mathtt{If0}\,x)\,M)\,N$. So $[\![M\,\mathtt{or}\,N]\!] = [\![M]\!] \sqcap [\![N]\!]$.

We now define the retraction from $U \Rightarrow \Sigma$ to $\mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow (\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma$.

Definition 10. $\mathtt{in} : ((\mathtt{nat} \Rightarrow \mathtt{o} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o}) \Rightarrow \mathtt{nat} \Rightarrow \mathtt{nat} \Rightarrow (\mathtt{nat} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o} =$

$$\mathtt{Y}\lambda F.\lambda f.\lambda x.((\lambda g.f\ \lambda u.\lambda v.(g\,u)) :: \langle (F\ \lambda z.f\ z[\lambda w.w]_x), F\ \lambda z.f\ z[\lambda w.\mathbf{e}]_x\rangle)$$

and $\mathtt{out} : (\mathtt{nat} \Rightarrow \mathtt{nat} \Rightarrow (\mathtt{nat} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o}) \Rightarrow (\mathtt{nat} \Rightarrow \mathtt{o} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o} =$

$$\mathtt{Y}\lambda G.\lambda h.\lambda k.(\mathtt{hd}\,(h\,0))\ \lambda a.(((k\,a)\ (G\ (\pi_0\ (\mathtt{tl}\ (h\,a))))\ k)\mathtt{or}(G\ (\pi_1\ (\mathtt{tl}\ (h\,a)))))\ k)$$

We prove that these terms do indeed define a retraction by an ordinal induction on the unfolding of the fixpoints. For this we require a measure on $f \in U \Rightarrow \Sigma$ of the number of unfoldings required to compute $\mathsf{in}(f)$.

Definition 11. *For each ordinal λ we define the set of λ-*dependent *elements of $U \Rightarrow \Sigma$ inductively, as follows:*

f is λ-dependent if for all n such that f is n-strict there exists $\kappa < \lambda$ such that $\lambda x.f\,x[\lambda w.w]_n$ and $\lambda x.f\,x[\lambda w.\top]_n$ are κ-dependent.

Proposition 6. *If f is λ-dependent then $\mathsf{out}(\mathsf{in}(f)) = f$.*

Proof. By induction on λ. Unfolding the definition of out, we have $\mathsf{out}(\mathsf{in}(f))(d) = (\mathsf{hd}\,(\mathsf{in}(f)\,0))\,\lambda a.(((d\,a)\,(\mathsf{out}(\pi_0\,(\mathsf{tl}\,(\mathsf{in}(f)\,a)))(d))) \sqcap (\mathsf{out}(\pi_1\,(\mathsf{tl}\,(\mathsf{in}(f)\,a)))(d)))$.

Unfolding the recursive definition of in, we have: $\mathsf{in}(f) = \lambda x.((\lambda g.f\,\lambda u.\lambda v.(g\,u)) :: \langle \mathsf{in}(\lambda z.f\,z[\lambda w.w]_x), \mathsf{in}(\lambda z.f\,z[\lambda w.\top]_x\rangle))$.

So $\pi_0(\mathsf{tl}\,(\mathsf{in}(f)\,n)) = \mathsf{in}(\lambda z.f\,z[I]_n)$ and $\pi_1(\mathsf{tl}\,(\mathsf{in}(f)\,n)) = \mathsf{in}(\lambda z.f\,z[\top]_n)$, and $(\mathsf{hd}\,(\mathsf{in}(f)\,0))(e) = f\,\lambda u.\lambda v.(e\,u)$. Since $(\lambda u.\lambda v.(e\,u))(n)(\bot) = (\lambda u.\lambda v.(e\,u))(n)(\top)$ for all n, by Lemma 11 $(\mathsf{hd}\,(\mathsf{in}(f)\,0))(e) = f\,\lambda u.\lambda v.(e\,u) = \bigsqcap\{e(n) \mid n \in \mathsf{strict}(f)\}$.

Substituting these into the expansion of $\mathsf{out}(\mathsf{in}(f))(d)$, we have: $\mathsf{out}(\mathsf{in}(f))(d) = \bigsqcap\{(d(n)\,\mathsf{out}(\mathsf{in}(\lambda z.f\,z[I]_n))(d)) \sqcap \mathsf{out}(\mathsf{in}(\lambda f\,z[\top]_n))(d) \mid n \in \mathsf{strict}(f)\}$.

If $n \in \mathsf{strict}(f)$, then since f is λ-dependent, $\lambda z.f\,z[I]_n$ and $\lambda z.f\,z[\top]_n$ are κ-dependent for some $\kappa < \lambda$ and so by induction hypothesis, $\mathsf{out}(\mathsf{in}(\lambda z.f\,z[I]_n) = \lambda z.f\,z[I]_n$ and $\mathsf{out}(\mathsf{in}(\lambda z.f\,z[\top]_n)) = \lambda z.f\,z[\top]_n$.

So, as required, $\mathsf{out}(\mathsf{in}(f))(d) = \bigsqcap\{(d(n)\,f(d[I]_n) \sqcap f(d[\top]_n) \mid n \in \mathsf{strict}(f)\} = \bigsqcap\{f_n(d) \mid n \in \mathsf{strict}(f)\} = f(d)$ by Lemma 10.

Proposition 7. *Every function $f \in U \Rightarrow \Sigma$ is λ-dependent for some λ.*

Proof. Say that f is n-constant if $f(e[\bot]_n) = f(e[\top]_n)$ for all e. We show by induction on λ that for each $f \in U \Rightarrow \Sigma$ which is not λ-dependent, we can construct a sequence of distinct values $\langle n_\kappa(f) \mid \kappa \leq \lambda\rangle$ such that f is not $n_\kappa(f)$-constant for each $\kappa \leq \lambda$. Since the cardinality of such a sequence must be countable, f must be λ-dependent for some countable λ.

For the induction case, suppose f is not λ-dependent. Then for some $m \in \mathbb{N}_*$ such that f is m-strict, and for some $C \in \{I, \top\}$, $\lambda x.f\,x[C]_m$ is not κ-dependent for all $\kappa < \lambda$. Then $m \neq n_\kappa(\lambda x.f\,x[C]_m)$ for $\kappa < \lambda$, as $\lambda x.f\,x[C]_m$ is m-constant by definition. If f is n-constant, then so is $\lambda x.f\,x[C]_m$, and so f is not $n_\kappa(\lambda x.f\,x[C]_m)$-constant for any $\kappa < \lambda$. Hence we may define $n_\lambda(f) = m$ and $n_\kappa(f) = n_\kappa(\lambda x.f\,x[C]_m)$ for $\kappa < \lambda$.

Combining Propositions 6 and 7, we have shown:

Proposition 8. $U \Rightarrow \Sigma \trianglelefteq \mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow (\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma$

By composing definable retractions we may now show that U is a (definably) "reflexive object" (i.e. $U \Rightarrow U$ is a definable retract of U).

Proposition 9. $U \Rightarrow U \trianglelefteq U$.

Proof. We have $U \Rightarrow U \cong \mathbb{N}_* \Rightarrow \Sigma \Rightarrow (U \Rightarrow \Sigma)$
$\trianglelefteq \mathbb{N}_* \Rightarrow \Sigma \Rightarrow (\mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow (\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma)$ (by Proposition 8)
$\cong \mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow (\Sigma \Rightarrow (\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma)$
$\trianglelefteq \mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow ((\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma)$ (by Lemma 7)
$\trianglelefteq \mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow \mathbb{N}_* \Rightarrow U$ by (Lemma 8)
$\trianglelefteq U$.

Corollary 2. *For every pointed type-object P, $P \trianglelefteq U$.*

Proof. By structural induction on P. Clearly $\Sigma \trianglelefteq U$: if $P = \mathbb{N}_* \Rightarrow Q$ then $P \trianglelefteq \mathbb{N}_* \Rightarrow U \trianglelefteq U$, and if $P = Q_1 \Rightarrow Q_2$ then $P \trianglelefteq U \Rightarrow U \trianglelefteq U$.

Thus we have extended inequational completeness to all types and proved full abstraction.

Theorem 1. *For all terms M, N, $M \lesssim^{may} N$ if and only if $[\![M]\!]_{may} \sqsubseteq [\![N]\!]_{may}$ and $M \lesssim^{must} N$ if and only if $[\![M]\!]_{must} \sqsubseteq [\![N]\!]_{must}$.*

6 Conclusions and Further Directions

For the purposes of exposition we have restricted our attention to a very simple functional language, but bidomains have the potential to model a range of programming languages with non-deterministic features. As we have observed, one possible route to describing more expressive languages is via CPS interpretation. Alternatively, we may interpret lifted types such as call-by-value functions and lifted sums via the *powerdomain* monad: for a pre-bidomain $(D, \sqsubseteq, \leq)$, we define a complete bidomain $\mathcal{P}(D)$ as follows:

- elements are up-closed subsets of D, together with a least element $\bot$,
- $\sqsubseteq$ is the Smyth ordering — i.e. reverse inclusion, with $\bot \sqsubseteq X$ for all X,
- $\leq$ is the intersection of the Smyth and Egli-Milner orders: $X \leq Y$ if $X = \bot$ or $Y \subseteq X$ and for all $x \in X$ there exists $y \in Y$ such that $x \leq y$.

In general, using powerdomains to interpret lifting leads to models with "first-order" control operators (jumps) rather than all first-class continuations (we note that $\mathcal{P}(\mathbb{N}_*) \cong (\mathbb{N}_* \Rightarrow \Sigma) \Rightarrow \Sigma)$. It should also be possible to develop a semantics of recursive types in complete bidomains, based on limits of countable chains of approximants, as investigated in [5].

We have shown that the elements of our model are sequential functions: it would be interesting to relate them to *strategies* in game semantics, in which fully abstract models of functional-imperative languages with *bounded* non-determinism have been described [6]. Comparison with our extensional model may yield an approach to unbounded choice. (As we have suggested, we may interpret bounded choice using Berry's original notion of bidomain (which does require $\sqsubseteq$-continuity). As shown in [4], (Berry's) bidomains have a decomposition into a *bistructure* model of classical linear logic, yielding possible connections between models of concurrency and our semantics of bounded and unbounded non-determinism.

Acknowledgement

The author would like to thank Soren Lassen for comments on a previous version of this article.

References

1. K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.
2. G. Berry. Stable models of typed λ-calculi. In *Proceedings of the 5th International Colloquium on Automata, Languages and Programming*, number 62 in LNCS, pages 72–89. Springer, 1978.
3. R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *Proceedings of POPL '92*, 1992.
4. P.-L. Curien, G. Winskel, and G. Plotkin. Bistructures, bidomains and linear logic. In *Milner Festschrift*. MIT Press, 1997.
5. P. Di Gianantonio, F. Honsell, and G. Plotkin. Uncountable limits and the lambda calculus. *Nordic Journal of Computing*, 2(2):126 – 145, 1995.
6. R. Harmer and G. McCusker. A fully abstract games semantics for finite nondeterminism. In *Proceedings of the Fourteenth Annual Symposium on Logic in Computer Science, LICS '99*. IEEE Computer Society Press, 1998.
7. J. Laird. Bistability: an extensional characterization of sequentiality. In *Proceedings of CSL '03*, number 2803 in LNCS. Springer, 2003.
8. J. Laird. Sequentiality in bounded bidomains. *Fundamenta Informaticae*, 65:173 – 191, 2005.
9. S. B. Lassen and C. Pitcher. Similarity and bisimilarity for countable nondeterminism and higher-order functions. *Electronic Notes in Theoretical Computer Science*, 10, 1997.
10. P. B. Levy. Infinite trace equivalence. In *Games for Logic and Programming Languages*, pages 195 – 209, 2005.
11. J. Longley. Universal types and what they are good for. In *Domain Theory, Logic and Computation: Proceedings of the 2nd International Symposium on Domain Theory*. Kluwer, 2004.
12. G. Plotkin. Lectures on predomains and partial functions, 1985. Notes for a course given at the Center for the study of Language and Information, Stanford.

An Operational Characterization of Strong Normalization*

Luca Paolini[1], Elaine Pimentel[2], and Simona Ronchi Della Rocca[1]

[1] Dipartimento di Informatica, Università di Torino (Italy)
[2] Departamento de Matemática, Universidade Federal de Minas Gerais (Brazil)

Abstract. This paper introduces the Φ-calculus, a new call-by-value version of the λ-calculus, following the spirit of Plotkin's $\lambda\beta_v$-calculus. The Φ-calculus satisfies some interesting properties, in particular that its set of solvable terms coincides with the set of β-strongly normalizing terms in the classical λ-calculus.

1 Introduction

The standard λ-calculus equipped with the β-reduction is the paradigmatic language for the call-by-name functional computation. Its call-by-value version, historically called $\lambda\beta_v$-calculus, has been introduced by Plotkin in 1975 [11]. The $\lambda\beta_v$-calculus is based on a restriction of the β-rule, firing it only when the argument belongs to a particular subset of terms, called values. In [10, 13] two co-authors of this paper, in order to treat these two different calculi in a uniform way, introduced the $\lambda\Delta$-calculus, parametric with respect to a subset Δ of terms, called *input values*, which generalizes the idea of Plotkin's values. Some conditions on Δ have been stated assuring some good properties for the calculus, in particular confluence and standardization. These conditions are very natural: the set of input values must contain the set of variables and it must be closed under substitution and Δ-reduction (a further condition is necessary for having standardization). Note that, in this setting, the standard λ-calculus now can be seen as a degenerated case of a $\lambda\Delta$-calculus, where all terms are input values (that is, the set of input values is Λ). Plotkin's calculus coincides with the $\lambda\Gamma$-calculus, where $\Gamma = \mathrm{Var} \cup \{\lambda x.M \mid M \in \Lambda\}$.

The formalization of the $\lambda\Delta$-calculus, where both call-by-name and call-by-value calculus can be uniformly representable, is an useful tool for studying the relationship between these two notions of computations. Some interesting properties relating $\lambda\Lambda$ and $\lambda\Gamma$-calculus have been already proved. For example, it turns out that, in the $\lambda\Gamma$-calculus, the notion of normal form is meaningless since there are different Γ-normal forms that can be consistently equated (see [13]). But the notion of β-normal form has an important meaning also in this calculus: in [7] it was proved that two different $\beta\eta$-normal forms can be separated in the $\lambda\Gamma$-calculus, and hence they cannot be consistently equated in any model.

In [8] we further explored the relationship between β-normal forms and Γ-evaluation. In that paper, it was proved that the set of strongly β-normalizing

* Paper partially supported by MIUR-PRIN'04 FOLLIA Project and by CNPq.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 367–381, 2006.
© Springer-Verlag Berlin Heidelberg 2006

terms, with respect to a *lazy* reduction[1], coincides with the set of potentially Γ-valuable terms. A term is potentially Δ-valuable (where Δ is any set of input values) if and only if there is a substitution, replacing variables by closed input values, such that the substituted term reduces to an input value. Being input values the only terms that can be manipulated by the Δ-calculus (and that can be argument of a function) this class of terms is particularly interesting. For example, in Plotkin's operational semantics for the $\lambda\Gamma$-calculus [11] based on the SECD machine [6], a potentially Γ-valuable term is always different from a non potentially Γ-valuable term. Moreover, all the non potentially Γ-valuable terms are equated [13].

A first natural question that arises is if this analogy between a call-by-name strong normalization and a call-by-value evaluation can be further developed. In particular, we are interested to know if *there is a set of input values Φ such that the set of potentially Φ-valuable terms coincides with the set of strongly β-normalizing terms.*

The set of lazy strong β-normalizing terms and the set Γ of input values have an interesting structural analogy: in order to find a lazy β-normal form of a λ-term it is not necessary to reduce under a λ-abstraction. The same happens when checking if a λ-term is a Γ-input value: it is not necessary to look under a λ-abstraction since all λ-abstractions are in Γ. We will call *weak* any set of input values containing all λ-abstractions. It turns out that "to be weak" has some consequences. Extending the notion of solvability to a generic Δ-calculus in the natural way, a term M is called Δ-*solvable* if and only if there is a sequence $P_1, ..., P_n$ of Δ-input values such that $(\lambda\vec{x}.M)P_1...P_n =_\Delta \lambda x.x$ (where $\lambda\vec{x}.M$ is the term obtained from M by abstracting it with respect to all its free variables). In the $\lambda\Gamma$-calculus, the set of solvable terms is a proper subset of the set of potentially valuable terms. This is a consequence of the fact that Γ is weak: if U is a closed Γ-unsolvable term then $\lambda x.U \in \Gamma$ is a potentially valuable, but unsolvable term.

This yields to a second question: *is there a $\lambda\Delta$-calculus such that the set of Δ-potentially valuable terms coincides with the set of Δ-solvable terms?*

Certainly, such a Δ could not be weak. In fact, in order to check if a term is solvable, it is necessary to perform the evaluation *under* the λ-abstraction.

In this paper, we give an answer to the two questions posted above, presenting the $\lambda\Phi$-calculus, where Φ is a set of input values which is the minimal solution of a recursive equation. Φ is not weak, since it is a proper subset of the set of Φ-normal forms. It turns out that the $\lambda\Phi$-calculus enjoys, besides confluence, the standardization property. Moreover, Φ is a proper subset of the set of strongly β-normalizing terms, and we prove that Φ is minimal between all sets of terms answering the first question. Hence our result can be rephrased as: the whole set

[1] Following [13], lazy β-reduction is defined as the closure of the β-rule under *application*, but not under *abstraction*. This corresponds operationally to do not perform reduction under the λ-abstraction. In the field of real functional languages, "lazy" is used with a different meaning.

of strongly β-normalizing terms can be operationally described through a proper subset of it.

A further comment is in order. Plotkin's motivation on designing the $\lambda\Gamma$-calculus was to propose a paradigmatic language for the call-by-value evaluation in real programming languages and, from this point of view, the choice of a weak set of input values is natural for modeling the notion of closure in the sense of Landin [6]. On the other hand, our motivation is purely theoretical, and the $\lambda\Phi$-calculus presented here *is not* an alternative proposal for designing new call-by-value languages. In any case, implementing the Φ-calculus would be difficult, being the set Φ just semi-decidable. But we believe that this study is interesting by itself. In fact, Plotkin in [11], posed the question of an existence of call-by-value λ-language alternative to $\lambda\Gamma$. He said that the natural proposal was to choose the set of the β-normal forms as an alternative to Γ. Unfortunately, the set of β-normal forms induces a calculus lacking the confluence property, in fact β-normal forms are not input values in our sense. Hence the $\lambda\Phi$-calculus, enjoying both confluence and standardization, gives an answer to this further question as well.

The rest of the paper is organized as follows: Section 2 contains basic notions of the parametric $\lambda\Delta$-calculus; in Section 3 the $\lambda\Phi$-calculus is introduced and finally, in Section 4 the main theorem is stated and proved.

2 The Parametric λ-Calculus

A calculus is a language equipped with some reduction rules. We will consider here calculi sharing the same language, the language of λ-calculus, while they differ from each other in their reduction rules. In order to treat them in a uniform way we will use the notion of parametric calculus, the $\lambda\Delta$-calculus, that gives rise to different calculi by different instantiations of the parameter Δ. The $\lambda\Delta$-calculus has been studied in [10, 13]. We use the terminology of [2, 13].

Definition 1 (The language Λ). *Let* Var *be a countable set of variables. The set Λ of λ-terms is defined by the following grammar:*

$$M ::= x \mid MM \mid \lambda x.M$$

λ-terms will be ranged over by Latin capital letters. Sets of λ-terms will be denoted by Greek capital letters. If Θ denotes a set of terms $(\Theta)^0$ is the set of closed terms belonging to Θ.

Sometimes, we will refer to λ-terms simply as terms. As usual, terms will be considered modulo α-conversion, i.e., modulo names of bound variables. The symbol $\equiv$ will denote syntactical identity of terms, up to α-equivalence.

We will use the following abbreviations, in order to avoid an excessive number of parentheses, thereby $\lambda x_1 ... x_n.M$ will stand for $(\lambda x_1(...(\lambda x_n.M)...))$ and $MN_1N_2...N_n$ will stand for $(...((MN_1)N_2)...N_n)$. Moreover $\vec{M}$ will denote a sequence of terms $M_1,\ldots,M_n$, for some $n \geq 0$, and $\lambda\vec{x}.M$ and $\vec{M}\vec{N}$, will denote

respectively $\lambda x_1 \ldots x_n.M$ and $M_1 \ldots M_m N_1 \ldots N_n$, for some $n, m \geq 0$. The length of the sequence $\vec{N}$ is denoted by $\|\vec{N}\|$.

The $\lambda\Delta$-calculus consists of the language Λ equipped with a set $\Delta \subseteq \Lambda$ of input values, satisfying some closure conditions. Informally, input values represent already evaluated terms, that can be passed as arguments. The set Δ of input values and the reduction $\rightarrow_\Delta$, induced by it, are defined below.

Definition 2. *Let $\Delta \subseteq \Lambda$.*

(i) *The* Δ-reduction *($\rightarrow_\Delta$) is the contextual closure of the following rule:*

$$(\lambda x.M)N \rightarrow M[N/x] \text{ if and only if } N \in \Delta.$$

$(\lambda x.M)N$ *is a* Δ-redex *(or simply redex).*

(ii) $\rightarrow^+_\Delta$, $\rightarrow^*_\Delta$ *and* $=_\Delta$ *are respectively the transitive closure of $\rightarrow_\Delta$, the reflexive and transitive closure of $\rightarrow_\Delta$ and the symmetric, reflexive and transitive closure of $\rightarrow_\Delta$.*

(iii) *A set $\Delta \subseteq \Lambda$ is a* set of input values*, when the following conditions are satisfied:*
- $\mathrm{Var} \subseteq \Delta$ (Var-closure);
- $P, Q \in \Delta$ *implies* $P[Q/x] \in \Delta$*, for each* $x \in \mathrm{Var}$ (substitution closure);
- $M \in \Delta$ *and* $M \rightarrow_\Delta N$ *imply* $N \in \Delta$ (reduction closure).

The closure conditions on the set of input values assure us that the $\lambda\Delta$-calculus enjoys the confluence property for every Δ , i.e., the following theorem holds.

Theorem 3 (Confluence). *[10, 13] Let $M \rightarrow^*_\Delta N_1$ and $M \rightarrow^*_\Delta N_2$. There is Q such that both $N_1 \rightarrow^*_\Delta Q$ and $N_2 \rightarrow^*_\Delta Q$.*

Two particular instantiations of Δ give rise to the call-by-name and the call-by-value λ-calculus. The call-by-name λ-calculus (i.e., the standard λ-calculus equipped with the β-reduction) coincides with the $\lambda\Lambda$-calculus. The call-by-value λ-calculus (defined by Plotkin in [11]) coincides with the $\lambda\Gamma$-calculus, where $\Gamma = \mathrm{Var} \cup \{\lambda x.M \mid M \in \Lambda\}$.

Let Δ be a set of input values. A term of the $\lambda\Delta$-calculus is in *Δ-normal form* if and only if it does no contain occurrences of Δ-redexes. A term M is *strongly Δ-normalizing* if both M has a Δ-normal form and every reduction sequence starting from M eventually stops.

The set Δ-NF of Δ-normal forms can be defined in the following recursive way:

$$\begin{aligned}\Delta\text{-NF} = \mathrm{Var} &\cup \{xM_1 ... M_n \mid M_k \in \Delta\text{-NF } (1 \leq k \leq n)\} \\ &\cup \{\lambda \vec{x}.M \mid M \in \Delta\text{-NF}\} \\ &\cup \{(\lambda x.P)QM_1 ... M_n \mid P, Q, M_k \in \Delta\text{-NF}, Q \notin \Delta \quad (1 \leq k \leq n)\}.\end{aligned}$$

Note that for the $\lambda\Lambda$-calculus, being Λ its set of input values, the last case cannot happen, i.e., there are no normal forms of the shape $(\lambda x.P)QM_1 ... M_n$, so Λ-NF$\subseteq \Delta$-NF, for all Δ.

In the $\lambda\Gamma$-calculus, the notion of normal form is meaningless. In fact, there are different Γ-normal forms that can be consistently equated. The key notion, in a call by value setting, is the one of (potential) valuability, given in the next definition (see [9],[13]).

Definition 4. (i) *A term M is* Δ-valuable *if and only if there is $N \in \Delta$ such that $M \to^*_\Delta N$.*
(ii) *A term M is* potentially Δ-valuable *if and only if there is a substitution* $\mathbf{s}$*, replacing variables by closed terms belonging to Δ, such that* $\mathbf{s}(M)$ *is Δ-valuable.*

It is immediate to verify that a closed term is potentially Δ-valuable if and only if it is Δ-valuable. Note that the notion of Δ-normal form and that one of potentially Δ-valuable are orthogonal. As an example, consider the $\lambda\Gamma$-calculus, and the term $M \equiv (\lambda z.D)(yI)D$, where $I \equiv \lambda x.x$ and $D \equiv (\lambda z.zz)$. M is in Γ-normal form, but it is neither an input value nor potentially Γ-valuable. In fact, consider $M[Q/y]$, for some $Q \in (\Gamma)^0$. If QI reduces to an element in Γ then $M[Q/y] \equiv (\lambda z.D)(QI)D$ reduces to DD, which is not an input value. Otherwise $M[Q/y] \to^*_\Gamma (\lambda z.D)Q'D$, for every Q' such that $QI \to^*_\Gamma Q'$, which is not an input value. Thus $(\lambda z.D)(QI)D$ is not Γ-valuable. We call *Δ-liar-normal forms* terms which are in Δ-normal form but that are not potentially Δ-valuable.

In the $\lambda\Lambda$-calculus, the notion of solvability plays an important role, since in some sense the solvable terms represent the meaningful computations [2]. In [9], Γ-solvable and potentially Γ-valuable terms has been characterized. This notion has been extended to the parametric $\lambda\Delta$-calculus in [13].

Definition 5. (i) *A context $C[.]$ is Δ-valuable if and only if $C[.] \equiv (\lambda\vec{x}.[.])\vec{P}$ where each $P \in \vec{P}$ is such that $P \in \Delta$.*
(ii) *A term M is* Δ-solvable *if and only if there is a Δ-valuable context $C[.]$ such that:*

$$C[M] =_\Delta I.$$

(iii) *A term is* Δ-unsolvable *if and only if it is not Δ-solvable.*

Note that $(\lambda\vec{x}.M)\vec{N} =_\Delta I$ means $(\lambda\vec{x}.M)\vec{N} \to^*_\Delta I$, since I is in Δ-NF, for every Δ.

3 The $\lambda\Phi$-Calculus

As observed in the introduction, we are interested to know if there is a set of input values Φ such that the set of potentially Φ-valuable terms coincides with the set of strongly β-normalizing terms. Such a set cannot be weak i.e., it cannot contain all λ-abstractions. Since input values represent already evaluated terms, the natural choice would be to take Δ such that Δ coincides with its set of normal forms (i.e. $\Delta = \Delta$-NF). From the recursive definition of Δ-normal form

restricted to the case where Δ is a set of input values, the following equations would be obtained:

(1.1) $\Delta = \text{Var} \cup \{xM_1...M_n \mid M_k \in \Delta\ (1 \leq k \leq n)\} \cup \{\lambda\vec{x}.M \mid M \in \Delta\} \cup$
$\{(\lambda x.P)QM_1...M_n \mid P, Q, M_k \in \Delta\ (1 \leq k \leq n), Q \notin \Delta\}$;
(1.2) $M, P \in \Delta$ implies $M[P/x] \in \Delta$.

Note that the reduction closure for Δ is trivially satisfied, since we asked that terms in Δ are Δ-normal forms.

The only solution to the equation (1.1) is $\Delta = \Lambda\text{-}NF$. In fact, the set $\{(\lambda x.P)QM_1...M_n \mid P, Q, M_k \in \Delta\ (1 \leq k \leq n), Q \notin \Delta\}$ is empty, due to the contradictory condition on Q. Unfortunately, $\Delta = \Lambda\text{-}NF$ does not satisfy the equation (1.2).

On the other hand, the simpler way of forcing Δ to satisfy equation (1.2) would be to restrict Δ so that it contains, besides variables, only closed terms. That is, to choose $\Delta^\dagger = (\Lambda\text{-}NF)^0 \cup Var$. But $\Delta^\dagger$ does not solve our problem. In fact, the term $I(\lambda x.I(xx))$ is a strongly Λ-normalizing term, but it is not $\Delta^\dagger$-solvable. Indeed $I(\lambda x.I(xx))$ is a $\Delta^\dagger$-NF which cannot be reduced (in the $\lambda\Delta^\dagger$-calculus), since $\lambda x.I(xx) \notin \Delta^\dagger$.

The discussion above suggests that we should look for a set Δ which is a proper subset of the Δ-normal forms, and a proper superset of $(\Lambda\text{-}NF)^0 \cup Var$. Let us maintain the choice that Δ contains, besides variables, only closed terms. The previous pair of equations now become:

(2.1) $\Theta = \text{Var} \cup \{xM_1...M_n \mid M_k \in \Theta\ (1 \leq k \leq n)\} \cup \{\lambda\vec{x}.M \mid M \in \Theta\} \cup$
$\{(\lambda x.P)QM_1...M_n \mid P, Q, M_1...M_n \in \Theta\ Q \notin \Delta\}$;
(2.2) $\Delta = Var \cup (\Theta)^0$.

Note that $\Theta = \Delta$-NF and Δ is a set of input values.

But the last condition on equation (2.1) is too weak again, since now the set Θ may contain some Δ-liar-normal forms. As an example, $M \equiv (\lambda z.D)(yI)D$, where $D \equiv (\lambda z.zz)$, is a Δ-liar-normal form satisfying both the previous equations. Since Δ-liar-normal forms are Δ-unsolvable, such a set cannot supply an answer to our second question, i.e., cannot be such that the set of Δ-solvable terms coincide with the set of Δ-potentially valuable terms.

The following easy to prove property will help us on excluding such dangerous terms.

Property 6. If there is a substitution $\mathbf{s}$ such that $\mathbf{s}(P[Q/x]\vec{M}) \to^*_\Delta R \in \Delta$ and $\mathbf{s}(Q) \to^*_\Delta Q' \in \Delta$ then $\mathbf{s}((\lambda x.P)Q\vec{M}) \to^*_\Delta R$.

Taking into account the previous property, the pair of equations now becomes:

(3.1) $\Theta = \text{Var} \cup \{xM_1...M_n \mid M_k \in \Theta\ (1 \leq k \leq n)\} \cup \{\lambda\vec{x}.M \mid M \in \Theta\} \cup$
$\{(\lambda x.P)QM_1..M_n \mid Q, M_1, ..., M_n \in \Theta, Q \notin \Delta, P[Q/x]M_1..M_n \to^*{}_\Delta R \in \Theta\}$;
(3.2) $\Delta = Var \cup (\Theta)^0$.

The minimal solution of this pair of recursive equations is defined next.

Definition 7. *The sets of* λ*-terms* Υ_i, Φ_i $(i \in \mathbb{N})$ *are defined by mutual induction, as follows*

$$\begin{array}{ll} \Upsilon_0 & = \text{Var} \\ \Phi_i & = \text{Var} \cup (\Upsilon_i)^0 \\ \Upsilon_{i+1} & = \text{Var} \cup \{xM_1...M_n \mid M_k \in \Upsilon_i (1 \leq k \leq n)\} \cup \{\lambda\vec{x}.M \mid M \in \Upsilon_i\} \\ & \bigcup \left\{ (\lambda x.P)QM_1...M_n \;\middle|\; \begin{array}{l} Q \in \Upsilon_i - (\Lambda^0 \cup \text{Var}), \quad M_1, ..., M_n \in \Upsilon_i, \\ P[Q/x]M_1 \ldots M_n \to^*_{\Phi_i} R \in \Upsilon_i \end{array} \right\} \end{array}$$

Moreover, we define $\Upsilon = \cup_i \Upsilon_i$ *and* $\Phi = \text{Var} \cup (\Upsilon)^0$*.*

For example, $\Phi_0 = \text{Var}$, $\Upsilon_1 = \text{Var} \cup \{xy_1...y_n \mid y_i \in Var\} \cup \{\lambda\vec{x}.y \mid y \in \text{Var}\}$ and $\Phi_1 = \text{Var} \cup \{\lambda x_1...x_m.x_j \mid 1 \leq j \leq m\}$.

The following result holds trivially:

Lemma 8. (i) *For all* $i \in \mathbb{N}$*,* Φ_i *is a set of input value.*
(ii) $\Phi = \cup_i \Phi_i$*.*
(iii) Φ *is a set of input values.*
(iv) *For all* $i \in \mathbb{N}$*,* Υ *and* Υ_i *are not sets of input values.*

Also, it is easy to check that the following properties hold.

Property 9. (i) $\Upsilon_i \subset \Upsilon_{i+1}$, $\Phi_i \subset \Phi_{i+1}$ and $\to_{\Phi_i} \subseteq \to_{\Phi_{i+1}}$, for all $i \in \mathbb{N}$;
(ii) $\Upsilon_i \subset \Upsilon$, $\Phi_i \subset \Phi$ and $\to_{\Phi_i} \subseteq \to_{\Phi}$, for all $i \in \mathbb{N}$;
(iii) $\Upsilon^0 = \Phi^0$;
(iv) $M \in \Phi^0$ implies $M \equiv \lambda z.P$, for some $z \in \text{Var}$ and $P \in \Upsilon$ (note that $\Phi \subseteq \Gamma$);
(v) $\Phi \subseteq \Upsilon$ and $\Upsilon \subseteq \Phi$-NF;
(vi) Φ-NF $\not\subseteq \Upsilon$ and Φ-NF $\not\subseteq \Phi$.

Proof. (vi) Let $M \equiv \lambda z.(\lambda x.D)(zI)D$. $M \in \Phi$-NF since $zI \notin \Phi$. But $M \notin \Upsilon$ and $M \notin \Phi$. □

Lemma 8.(iii) implies that the $\lambda\Phi$-calculus enjoys the confluence property. Moreover it is possible to check that it also satisfies the additional necessary condition for standardization, stated in [10, 13].

4 The Main Result

The $\lambda\Phi$-calculus, besides confluence and standardization, has some further interesting properties. The most important one is that the set of potentially Φ-valuable terms coincides with the set of strongly Λ-normalizing terms. Other properties characterize completely the operational behavior of the $\lambda\Phi$-calculus. In particular, a term is Φ-solvable if and only if it is potentially Φ-valuable if and only if it Φ-reduces to a term in Υ.

Theorem 10 (Main Theorem). *The following statements are equivalent:*

(i) M *is strongly* Λ*-normalizing;*
(ii) M *is* Φ*-solvable;*
(iii) M *is potentially* Φ*-valuable;*
(iv) $M \to^*_{\Phi} R \in \Upsilon$*;*

Proof. (i) implies (ii) by Theorem 26.
(ii) implies (iii) by Theorem 16.(ii).
(iii) implies (iv) by Theorem 16.(i).
(iv) implies (i) by Theorem 20.
where theorems 16, 20 and 26 will be proved later in this Section. □

Notice that Φ is a *proper* subset of the set of strongly Λ-normalizing terms. In fact, a strongly Λ-normalizing term of the shape $M_1...M_2$ where M_i is closed $(1 \leq i \leq n)$ does not belong to Φ. Moreover Φ is minimal between all sets of input values which answer our first question.

Property 11. Let $\Delta^\star$ be a set of input values.
If the set of potentially $\Delta^\star$-valuable terms coincides with the set of the strongly Λ-normalizing terms and $\Delta^\star \subset \Phi$ then $\Delta^\star = \Phi$.

Proof. Clearly $\Delta^\star = \Phi$ if and only if $(\Delta^\star)^0 = \Phi^0$, since $\Delta^\star \subset \Phi$ and $\Phi = \mathrm{Var} \cup \Phi^0$. Thus suppose $M \in \Phi^0$. Note that M is Φ-valuable, potentially Φ-valuable and also in Φ-normal form. Moreover M is a closed strongly Λ-normalizing term, by the Main Theorem. Thus, M is potentially $\Delta^\star$-valuable by hypothesis and $M \in \Lambda^0$ implies that M is $\Delta^\star$-valuable. But $\Delta^\star \subset \Phi$ implies Φ-NF $\subset \Delta^\star$-NF, hence $M \in \Delta^\star$-NF. Then $M \in \Delta^\star$ and the proof is done. □

Finally, it is worthy to say that, although Φ is minimal, it is not *the* minimum set supplying an answer to questions stated in the introduction. In fact, the minimum solution to the following equations:

$$\begin{aligned}\Theta = &\{\lambda x_0...x_n.y \mid y \neq x_i \ \ (0 \leq i \leq n)\ \} \cup \\ &\{xM_1...M_n \mid M_k \in \Theta\ (1 \leq k \leq n)\} \cup \{\lambda \vec{x}.M \mid M \in \Theta\} \cup \\ &\{(\lambda x.P)QM_1...M_n \,|\, Q, M_1,..., M_n \in \Theta,\ Q \notin \Delta,\ P[Q/x]M_1..M_n \rightarrow^*{}_{\Delta} R \in \Theta\} \\ \Delta = &\{\lambda x_0...x_n.y \mid y \neq x_i \ \ (0 \leq i \leq n)\ \} \cup (\Theta)^0\end{aligned}$$

also answers our questions and it is not comparable with Φ.
The rest of the paper is devoted to the proof of the Main Theorem.

4.1 Potential Φ-Valuability and Φ-Solvability

First of all, we will introduce the *weight* of a terms, as measure for carrying out some inductive proofs. The weight of a term M is an upper bound to the number of symbols of M, to the length of its leftmost Λ-reduction sequence and to the length of its Φ-reduction sequence according to the standard strategy [10].

Definition 12. *The* weight $\langle _ \rangle : \Lambda \longrightarrow \mathbb{N}$ *is the partial function defined as follows:*

- $\langle \lambda x.M' \rangle = 1 + \langle M' \rangle$.
- $\langle xM_1...M_m \rangle = 1 + \langle M_1 \rangle + ... + \langle M_m \rangle$.
- $\langle (\lambda x.M_0)M_1...M_m \rangle = 1 + \langle M_1 \rangle + \langle M_0[M_1/x]M_2...M_m \rangle$.

As examples $\langle x \rangle = 1$, $\langle xx \rangle = 2$, $\langle \lambda x.xx \rangle = 3$. It is easy to check that $M \rightarrow_\Phi N$ implies $M[P/z] \rightarrow_\Phi N[P/z]$, if $P \in \Phi$.

Lemma 13. (i) *If* $M \in \Upsilon$ *then* $\langle M \rangle \in \mathbb{N}$.
(ii) *If* $M \rightarrow^+_\Phi N$ *and* $\langle N \rangle \in \mathbb{N}$ *then, both* $\langle M \rangle \in \mathbb{N}$ *and* $\langle N \rangle < \langle M \rangle$.
(iii) *If* $M \rightarrow^+_\Phi N$ *and* $\langle M \rangle \in \mathbb{N}$ *then, both* $\langle N \rangle \in \mathbb{N}$ *and* $\langle N \rangle < \langle M \rangle$.

Proof. (i) The proof can be done by induction on the Υ stratification.
(ii) The proof is given by induction on $\langle N \rangle$.
- If $M \equiv \lambda x.P \rightarrow^+_\Phi \lambda x.P' \equiv N$ then $\langle P' \rangle \in \mathbb{N}$ implies $\langle P' \rangle < \langle P \rangle \in \mathbb{N}$ by induction, hence $\langle N \rangle < 1 + \langle P \rangle = \langle M \rangle$.
- Let $M \equiv xM_1...M_m \rightarrow^+_\Phi xN_1...N_m \equiv N$ $(m \geq 1)$ where either $M_k \rightarrow^+_\Phi N_k$ or $M_k \equiv N_k$ $(1 \leq k \leq m)$. Note that there is at least one $h \in \mathbb{N}$ such that $M_h \rightarrow^+_\Phi N_h$ and $\langle N_h \rangle < \langle M_h \rangle$ $(1 \leq h, k \leq m)$. Thus the proof follows easily by induction.
- Let $M \equiv (\lambda z.M_0)M_1...M_m \rightarrow^+_\Phi N$ for some $m \geq 1$.
  Either $M \rightarrow^*_\Phi (\lambda x.N_0)N_1...N_m \rightarrow_\Phi N_0[N_1/x]N_1...N_m \rightarrow^*_\Phi N$ or $M \rightarrow^+_\Phi (\lambda x.N_0)N_1...N_m \equiv N$, where $M_k \rightarrow^+_\Phi N_k$ or $M_k \equiv N_k$ $(1 \leq k \leq m)$. In all cases, the proof follows by induction.

(iii) The proof is given by induction on $\langle M \rangle$.
- If $M \equiv \lambda x.M_0 \rightarrow^+_\Phi \lambda x.N_0 \equiv N$ then $\langle M_0 \rangle \in \mathbb{N}$. Hence $\langle N_0 \rangle < \langle M_0 \rangle$ by induction, thus $\langle N \rangle = 1 + \langle N_0 \rangle < 1 + \langle M_0 \rangle = \langle M \rangle$.
- Let $M \equiv xM_1...M_m \rightarrow^+_\Phi xN_1...N_m \equiv N$ $(m \geq 1)$, where either $M_k \rightarrow^+_\Phi N_k$ or $M_k \equiv N_k$ $(1 \leq k \leq m)$. Note that there is $h \in \mathbb{N}$ such that $M_h \rightarrow^+_\Phi N_h$ and $\langle N_h \rangle < \langle M_h \rangle$. Thus the proof follows easily by induction.
- Let $M \equiv (\lambda z.M_0)M_1...M_m \rightarrow^+_\Phi N$ $(m \geq 1)$.
  Either $M \rightarrow^*_\Phi (\lambda x.N_0)N_1...N_m \rightarrow_\Phi N_0[N_{1/x}]N_1...N_m \rightarrow^*_\Phi N$ or $M \rightarrow^+_\Phi (\lambda x.N_0)N_1...N_m \equiv N$, where $M_k \rightarrow^+_\Phi N_k$ or $M_k \equiv N_k$ $(1 \leq k \leq m)$.
  If $M_1 \rightarrow^+_\Phi N_1$ then $\langle N_1 \rangle < \langle M_1 \rangle$ and the proof follows by induction.
  Otherwise $M_1 \equiv N_1$ and $M_0[M_1/x]M_2...M_m \rightarrow^+_\Phi N_0[M_1/x]N_2...N_m$, hence $\langle M_0[M_1/x]M_2...M_m \rangle < \langle N_0[M_1/x]N_2...N_m \rangle$ and the proof follows immediately, in all cases. □

It is possible to characterize the terms for which the weight is defined.

Corollary 14. $\langle M \rangle$ *is defined if and only if* $M \rightarrow^*_\Phi R$*, for some* $R \in \Upsilon$.

Proof. $\Leftarrow$ The proof follows by Lemma 13.(i) and Lemma 13.(ii).
$\Rightarrow$ The proof is given by induction on $\langle M \rangle$.
If $M \equiv \lambda x.M_0$ or $M \equiv xM_1...M_m$ $(m \in \mathbb{N})$ then the proof follows by induction. Let $M \equiv (\lambda z.M_0)M_1...M_m$ $(m \geq 1)$, so $\langle M_1 \rangle < \langle M \rangle$ and by induction $M_1 \rightarrow^*_\Phi S \in \Upsilon$.
- If $M \in \Phi$-NF then $M_1 \in \Upsilon$, but $M_1 \notin \Phi = \Upsilon^0 \cup \mathrm{Var}$. Furthermore $\langle M_0[M_1/x]M_2...M_m \rangle < 1 + \langle M_1 \rangle + \langle M_0[M_1/x]M_2...M_m \rangle = \langle M \rangle$ implies that $M_0[M_1/x]M_2...M_m \rightarrow^*_\Phi R'$, for some $R' \in \Upsilon$. The proof follows by definition of Υ.
- Otherwise $M \rightarrow^+_\Phi N$, so the proof follows by Lemma 13.(iii) and by induction. □

If the weight of a term is defined, then the weight of all its subterms is also defined. The next lemma proves this statement in some particular cases.

Lemma 15. (i) *If* $M[N/z] \to_{\Phi}^{*} R \in \Upsilon$ *then* $M \to_{\Phi}^{*} S \in \Upsilon$.
(ii) *If* $MN \to_{\Phi}^{*} R \in \Upsilon$ *then* $M \to_{\Phi}^{*} S \in \Upsilon$.

Proof. (i) We will prove that $\langle M[N/z] \rangle \in \mathbb{N}$ implies $\langle M \rangle \in \mathbb{N}$ by induction on $h = \langle M[N/z] \rangle$, thus the proof follows by Corollary 14.

Let $M \equiv \lambda x.M_0$. $\langle M_0[N/z] \rangle \in \mathbb{N}$ implies $\langle M_0 \rangle < h$ by induction and the proof follows. If $M \equiv xM_1...M_m$ $(m \geq 1)$ then, in all cases, $\langle M_i[N/z] \rangle < h$ $(1 \leq i \leq m)$ and the proof follows by induction. If $M \equiv (\lambda x.M_0)M_1...M_m$ $(m \geq 1)$ then $\langle M[N/z] \rangle = 1 + \langle M_1[N/z] \rangle + \langle (M_0[M_1/x]M_2...M_m)[N/z] \rangle$. Thus $\langle M_1 \rangle$ and $\langle M_0[M_1/x]M_2...M_m \rangle$ are defined by induction and the proof follows.

(ii) We will prove that $\langle MN \rangle \in \mathbb{N}$ implies $\langle M \rangle \in \mathbb{N}$ by induction on $\langle MN \rangle$, thus the proof follows by Corollary 14.

If $M \equiv \lambda x.M_0$ then $\langle MN \rangle = 1 + \langle N \rangle + \langle M_0[N/z] \rangle$, so $\langle M_0[N/z] \rangle \in \mathbb{N}$ and the proof follows by the previous point of this Theorem and the definition of weight. The other cases are easier. □

Theorem 16. (i) *M is potentially Φ-valuable implies that* $M \to_{\Phi}^{*} R \in \Upsilon$, *for some* $R \in \Upsilon$.
(ii) *M is Φ-solvable implies that M is potentially Φ-valuable.*

Proof. (i) M is potentially Φ-valuable means that there is a substitution $\mathbf{s}$, replacing variables by closed terms belonging to Φ, such that $\mathbf{s}(M) \to_{\Phi}^{*} N \in \Phi$. Since $\Phi \subseteq \Upsilon$, the proof follows by Lemma 15(i).

(ii) M is Φ-solvable means that there is a Φ-valuable context $C[.] \equiv (\lambda \vec{x}.[.])\vec{N}$ such that $C[M] \to_{\Phi}^{*} I$. Since $C[M] \to_{\Phi}^{*} I$ implies $C[M]I...I \to_{\Phi}^{*} I$, we can assume $\|\vec{x}\| \leq \|\vec{N}\|$ without loss of generality. Moreover $C[M] \to_{\Phi}^{*} I$ implies $C[M][N/z] \to_{\Phi}^{*} I \equiv I[N/z]$ for all $N \in \Phi^0$, so we can also assume that $C[M] \in \Lambda^0$.

If $\vec{N} \equiv \vec{N_0}\vec{N_1}$ and $\|\vec{x}\| = \|\vec{N_0}\|$ then $C[M] \to_{\Phi}^{*} M[\vec{N_0}/\vec{x}]\vec{N_1} \to_{\Phi}^{*} I \in \Phi$, thus $M[\vec{N_0}/\vec{x}] \to_{\Phi}^{*} S' \in \Upsilon^0$, by Lemma 15.(ii). The proof is done, since $\Upsilon^0 = \Phi^0$. □

4.2 Strong Λ-Normalization and Φ-Reduction

In order to prove both that the terms strongly Λ-normalizing are also Φ-solvable and that terms which Φ-reduce to an element of Υ are strongly Λ-normalizing, we will use an intersection type assignment system [1, 4] that types all and only the strongly Λ-normalizing terms [5, 12].

Definition 17. (i) *Let* C *be a countable set of* type-constants *(ranging over* $\alpha, \beta, ..$*). The set* $T(\mathsf{C})$ *of* types, *ranging over by* $\sigma, \tau, \pi, \rho, ..$ *is inductively defined as follows:*

$$\begin{array}{lcl} \sigma \in \mathsf{C} & \Rightarrow & \sigma \in T(\mathsf{C}) \\ \sigma, \tau \in T(\mathsf{C}) & \Rightarrow & (\sigma \to \tau) \in T(\mathsf{C}) \\ \sigma, \tau \in T(\mathsf{C}) & \Rightarrow & (\sigma \wedge \tau) \in T(\mathsf{C}). \end{array}$$

We use the convention that the constructor $\wedge$ *take precedence over* $\to$.

(ii) *A basis is a partial function from* Var *to* $T(\mathsf{C})$ *having a finite domain of definition. If* B *is a basis then* $B[\sigma/x]$ *denotes the basis such that*

$$B[\sigma/x](y) = \begin{cases} \sigma & \text{if } y \equiv x, \\ B(y) & \text{otherwise.} \end{cases}$$

Furthermore, the basis B *such that* $dom(B) = \{x_1, ..., x_n\}$ *and* $B(x_i) = \sigma_i$*, for* $1 \leq i \leq n$ *will be often denoted by* $[\sigma_1/x_1, ..., \sigma_n/x_n]$*.*

(iii) *The* type assignment system $\vdash$ *is a formal system proving typing judgments of the shape:*

$$B \vdash M : \sigma$$

where M *is a term,* $\sigma \in T(\mathsf{C})$ *and* B *is a basis.*

The type assignment system $\vdash$ *consists of the following rules:*

$$\frac{}{B[\sigma/x] \vdash x : \sigma}\ (var)$$

$$\frac{B[\sigma/x] \vdash M : \tau}{B \vdash \lambda x.M : \sigma \to \tau}\ (\to I) \qquad \frac{B \vdash M : \sigma \to \tau \quad B \vdash N : \sigma}{B \vdash MN : \tau}\ (\to E)$$

$$\frac{B \vdash M : \sigma \quad B \vdash M : \tau}{B \vdash M : \sigma \wedge \tau}\ (\wedge I)$$

$$\frac{B \vdash M : \sigma \wedge \tau}{B \vdash M : \sigma}\ (\wedge E_l) \qquad \frac{B \vdash M : \sigma \wedge \tau}{B \vdash M : \tau}\ (\wedge E_r)$$

If B, B' are bases then $B \cap B'$ is the basis defined as follows:

$$(B \cap B')(y) = \begin{cases} B(y) \wedge B'(y) & \text{if both } B(y) \text{ and } B'(y) \text{ are defined,} \\ B(y) & \text{if } B(y) \text{ is defined and } B'(y) \text{ is undefined,} \\ B'(y) & \text{if } B'(y) \text{ is defined and } B(y) \text{ is undefined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The type systems $\vdash$ enjoys the subject-reduction property and a restricted form of subject-expansion.

Property 18 (Subject-Reduction and Typed Subject-Expansion).

(i) If $B \vdash M : \sigma$ and $M \to_\Lambda N$ then $B \vdash N : \sigma$.
(ii) Let $C[.]$ be a context. Then $B \vdash C[P[Q/x]] : \sigma$ and $B' \vdash Q : \tau$ imply $B \cap B' \vdash C[(\lambda x.P)Q] : \sigma$.

Proof. See [4]. □

Lemma 19. *If $M \in \Upsilon$ then $B \vdash M : \sigma$, for some basis B and $\sigma \in T(\mathsf{C})$.*

Proof. The proof is given by induction on the stratification of Υ. The case $M \in \Upsilon_0$ is trivial. If $M \in \Upsilon_{i+1}$, the cases $M \equiv xM_1...M_m$ and $M \equiv \lambda x.P$ follow easily from the inductive hypothesis, using the rules of the system. If $M \equiv (\lambda x.P)QM_0...M_m$ then both $Q \in \Upsilon_i - (\Lambda^0 \cup \mathrm{Var})$ and $P[Q/x]M_1 \ldots M_m \to^*_{\Phi_i} R \in \Upsilon_i$. Therefore $B \vdash R : \sigma$, by induction. Since the $\to^*_{\Phi_i}$ reduction reduces only in case the argument belongs to Φ_i this can be typed by induction, and hence $B \vdash P[Q/x]M_1 \ldots M_m : \sigma$ by Property 18.(ii). Since $B' \vdash Q : \tau$ by induction, the proof follows by Property 18.(ii) . □

Theorem 20. *$M \to^*_{\Phi} R \in \Upsilon$ implies M is Λ-strongly normalizing.*

Proof. By Lemma 19, $B \vdash R : \sigma$ for some basis B and $\sigma \in T(\mathsf{C})$. Thus $B \vdash M : \sigma$ by Property 18.(ii). Then the proof follows from the fact that the system characterizes the strongly Λ-normalizing terms [5, 12]. □

4.3 Strong Λ-Normalization and Φ-Solvability

In order to prove that Λ-strong normalization implies Φ-solvability, we will use a computability argument, which is an adaptation to intersection types of the reducibility candidates method [3].

Let M be a term, let $\mathrm{FV}(M) \subseteq \{x_1, ..., x_m\}$ for some $m \geq 0$ and let O^i be $\lambda x_0...x_i.x_i$ for all $i \in \mathbb{N}$.

The meaning of $\mathcal{P}(M)$ will be: "there is $r \in \mathbb{N}$ such that, for all $h, k \geq r$,

$$M[O^h/x_1, ..., O^h/x_m]\underbrace{O^h.....O^h}_{k} \to^*_{\Phi} O^s \ \text{ for some } \ s \in \mathbb{N}."$$

Property 21. (i) $\mathcal{P}(M)$ implies M is Φ-solvable.
(ii) $\mathcal{P}(x\overrightarrow{M})$ and $\mathcal{P}(N)$ imply $\mathcal{P}(x\overrightarrow{M}N)$.

Proof. (i) $M[O^h/x_1, ..., O^h/x_m]\underbrace{O^h.....O^h}_{k} \to^*_{\Phi} O^s$ implies that

$$M[O^h/x_1, ..., O^h/x_m]\underbrace{O^h.....O^h}_{k}\underbrace{I.....I}_{s} \to^*_{\Phi} I.$$

(ii) Let $\overrightarrow{M} \equiv M_1...M_n$ $(n \in \mathbb{N})$ and let $\mathrm{FV}(x\overrightarrow{M}N) \subseteq \{x_1, ..., x_m\}$ $(m \in \mathbb{N})$.

$$\exists r_0 \in \mathbb{N},\ \forall h_0, k_0 \geq r_0,\ \exists s_0 \in \mathbb{N},\ x\overrightarrow{M}[O^{h_0}/x_1, ..., O^{h_0}/x_m]\underbrace{O^{h_0}.....O^{h_0}}_{k_0} \to^*_{\Phi} O^{s_0},$$

$$\exists r_1 \in \mathbb{N},\ \forall h_1, k_1 \geq r_1,\ \exists s_1 \in \mathbb{N},\ N[O^{h_1}/x_1, ..., O^{h_1}/x_m]\underbrace{O^{h_1}.....O^{h_1}}_{k_1} \to^*_{\Phi} O^{s_1},$$

by hypothesis. So the proof follows by putting $r = \max\{r_0, r_1, n+1\}$. □

The predicate $\mathcal{P}$ is used to define the computability predicate.

Definition 22. *The predicate Comp is defined by induction on types as follows:*

- $Comp(B, \alpha, M)$ *if and only if* $\mathcal{P}(M)$, $\alpha \in \mathsf{C}$ *and* B *is a basis;*
- $Comp(B, \sigma \to \tau, M)$ *if and only if, for all* $N \in \Lambda$, $Comp(B', \sigma, N)$ *implies* $Comp(B \cap B', \tau, MN)$*;*
- $Comp(B, \sigma \wedge \tau, M)$ *if and only if* $Comp(B, \sigma, M)$ *and* $Comp(B, \tau, M)$.

We prove that $B \vdash M : \sigma$ implies $Comp(B, \sigma, M)$, which in its turn implies $\mathcal{P}(M)$. It is easy to check that $Comp(B, \sigma, M)$ does not imply $B \vdash M : \sigma$.

Lemma 23. (i) $\mathcal{P}(x\overrightarrow{M})$ *implies* $Comp(B, \sigma, x\overrightarrow{M})$*, for all* B *and* $\sigma \in T(\mathsf{C})$.
(ii) $Comp(B, \sigma, M)$ *implies* $\mathcal{P}(M)$*, for all* B *and* $\sigma \in T(\mathsf{C})$.

Proof. The proof is by mutual induction on σ. The only case which is not obvious is when $\sigma = \tau \to \rho$.

(i) We will prove that $Comp(B', \tau, N)$ implies $Comp(B \cap B', \rho, x\overrightarrow{M}N)$, thus $Comp(B, \tau \to \rho, x\overrightarrow{M})$ follows by definition. $Comp(B', \tau, N)$ implies $\mathcal{P}(N)$, by induction. But $\mathcal{P}(x\overrightarrow{M})$ by hypothesis, thus $\mathcal{P}(x\overrightarrow{M}N)$ by Property 21.(ii).

(ii) $\mathcal{P}(z)$ holds so, in particular, $Comp(B', \tau, z)$ holds by induction on (i). Thus $Comp(B \cap B', \rho, Mz)$ by definition of $Comp$ and this implies $\mathcal{P}(Mz)$ by induction. That is, there is $r \in \mathbb{N}$ such that, for all $h, k \geq r$,

$$Mz[O^h/z, O^h/x_1, ..., O^h/x_m]\underbrace{O^h.....O^h}_{k} \to_{\Phi}^{*} O^s$$

where $s \in \mathbb{N}$, $O^h \equiv \lambda x_0...x_h.x_h$ and $\mathrm{FV}(Mz) \subseteq \{z, x_1, ..., x_m\}$ $(m \geq 0)$. So, if $r' = r + 1$, for all $h, k \geq r'$, $M[O^h/x_1, ..., O^h/x_m]\underbrace{O^h.....O^h}_{k} \to_{\Phi}^{*} O^s$, for some s. Hence $\mathcal{P}(M)$ holds. □

Property 24. Let Q be such that $\mathcal{P}(Q)$.
If $Comp(B, \sigma, P[Q/x]\overrightarrow{Q})$ then $Comp(B, \sigma, (\lambda x.P)Q\overrightarrow{Q})$.

Proof. The proof is by induction on the structure of types.
Assume $\sigma \in \mathsf{C}$. Note that $\mathcal{P}(Q)$ and Lemma 15.(i) imply

$$\exists r_0 \in \mathbb{N},\ \forall h_0, k_0 \geq r_0,\ \ Q[O^{h_0}/x_1, ..., O^{h_0}/x_m] \to_{\Phi}^{*} M \in \Upsilon^0 = \Phi^0.$$

Moreover, $Comp(B, \sigma, P[Q/x]\overrightarrow{Q})$ implies $\mathcal{P}(P[Q/x]\overrightarrow{Q})$ by definition, thus

$$\exists r_1,\ \forall h_1, k_1 \geq r_1,\ \exists s_1,\ (P[Q/x]\overrightarrow{Q})[O^{h_1}/x_1, ..., O^{h_1}/x_m]\underbrace{O^{h_1}.....O^{h_1}}_{k_1} \to_{\Phi}^{*} O^{s_1}.$$

Let $r = \max\{r_0, r_1\}$. Then, $\forall h, k \geq r$, $((\lambda x.P)Q\overrightarrow{Q})[O^h/x_1, ..., O^h/x_m]\underbrace{O^h.....O^h}_{k}$

$=_{\Phi} (P[Q/x]\overrightarrow{Q})[O^h/x_1, ..., O^h/x_m]\underbrace{O^h.....O^h}_{k} \to_{\Phi}^{*} O^s$, for some $s \in \mathbb{N}$.

Hence $\mathcal{P}(Q)$ and $\sigma \in \mathsf{C}$ imply $Comp(B, \sigma, (\lambda x.P)Q\overrightarrow{Q})$.

Let $\sigma = \tau \to \rho$. Thus $Comp(B, \tau \to \rho, P[Q/x]\overrightarrow{Q})$ implies that $\forall N$ such that $Comp(B', \tau, N)$, $Comp(B \cap B', \rho, P[Q/x]\overrightarrow{Q}N)$ holds. Therefore by induction, $Comp(B \cap B', \rho, (\lambda x.P)Q\overrightarrow{Q}N)$ and hence $Comp(B \cap B', \tau \to \rho, (\lambda x.P)Q\overrightarrow{Q})$ by definition of $Comp$. The case $\sigma = \tau \wedge \rho$ is trivial. □

Lemma 25. *Let* $\mathrm{FV}(M) \subseteq \{x_1, ..., x_n\}$ *and* $B(x_i) = \sigma_i$ $(1 \leq i \leq n)$*.*
If $Comp(B_i, \sigma_i, N_i)$ $(1 \leq i \leq n)$ *and* $B \vdash M : \tau$*, then*

$$Comp(B_1 \cap ... \cap B_n, \tau, M[N_1/x_1, ..., N_n/x_n]).$$

Proof. By induction on the derivation d of $B \vdash M : \tau$. The most interesting case is when the last rule applied on d is $(\to I)$. Let $M \equiv \lambda x.M'$, $\tau = \mu \to \rho$ and

$$\frac{B[\mu/x] \vdash M' : \rho}{B \vdash \lambda x.M' : \mu \to \rho}\,(\to I)$$

Suppose that $Comp(B', \mu, N)$ holds. Then $\mathcal{P}(N)$ holds by Lemma 23.(ii). Thus, by induction

$$Comp(B' \cap B_1 \cap ... \cap B_n, \rho, M'[N_1/x_1, ..., N_n/x_n, N/x])$$

and $Comp(B' \cap B_1 \cap ... \cap B_n, \rho, (\lambda x.M'[N_1/x_1, ..., N_n/x_n])N)$ by Property 24. Hence, $Comp(B_1 \cap ... \cap B_n, \mu \to \rho, M[N_1/x_1, ..., N_n/x_n])$ by definition of $Comp$. All other cases follow directly from the inductive hypothesis. □

Theorem 26. *M is Λ-strongly normalizing implies that M is Φ-solvable.*

Proof. In [12, 5] it is proved that the system $\vdash$ characterizes the strongly Λ-normalizing terms. So, let $B \vdash M : \sigma$, $\mathrm{FV}(M) \subseteq \{x_1, ..., x_n\}$ and $B(x_i) = \sigma_i$. Since $Comp(B, \sigma_i, x_i)$ $(1 \leq i \leq n)$ by Lemma 23.(i), then $Comp(B, \sigma, M)$ by Lemma 25. Thus $\mathcal{P}(M)$ by Lemma 23.(ii). The proof follows by Property 21.(i). □

References

1. van Bakel S., *Intersection type assignment systems*, Theoretical Computer Science, 38(2):246-269, Elsevier, 1997.
2. Barendregt H.P., *The Lambda Calculus: its syntax and semantics*, N.103 in Studies in Logic and the Foundations of Mathematics (revised edition), North-Holland, Amsterdam, 1994.
3. Coppo M., Dezani-Ciancaglini M., Zacchi M., *Type Theories, Normal Forms and $\mathcal{D}_\infty$ Lambda Models*, Information and Control, 72, 2, 1987, pp.85-116.
4. Coppo M., Dezani-Ciancaglini M., *An Extension of the Basic Functionality Theory for the λ-Calculus* Notre-Dame Journal of Formal Logic, 21(4), pp. 685-693, October 1980.
5. Krivine J.L., *Lambda-Calculus, Types and Models*, Ellis Horwood Series in Computers and Their Applications. 1993.

6. Landin P.J., *The mechanical evaluation of expressions*, Computer Journal, 1964.
7. Paolini L.,*Call-by-value separability and computability*, ICTCS'01, Restivo, Ronchi Della Rocca, and Roversi, eds, LNCS 2202, Springer-Verlag, 74-89.
8. Paolini L., Pimentel E., Ronchi Della Rocca S. *Lazy strong normalization*, ITRS'04, ENTCS vol. 136, pp. 103–116, 2005.
9. Paolini L., Ronchi Della Rocca S., *Call-by-value Solvability*, Theoretical Informatics and Applications, 33(6), 507-534, 1999.
10. Paolini L., Ronchi Della Rocca S., *The Parametric Parameter Passing λ-calculus*, Information and Computation, 189(1):87-106, 2004.
11. Plotkin G.D., *Call-by-name, call-by-value and the λ-calculus*, Theoretical Computer Science (1) 125-159, 1975.
12. Pottinger G., *A type assignment for the strongly normalizable λ-terms*, in To H.B. Curry: essays on combinatory logic, lambda calculus and formalism, pp.561-577, Academic Press, London, 1980.
13. Ronchi Della Rocca S., Paolini L., *The Parametric λ-calculus. A meta-model for computation*, Texts in Theoretical Computer Science: an EATCS Series, Springer-Verlag, Berlin, 2004.

On the Confluence of λ-Calculus with Conditional Rewriting

Frédéric Blanqui[1], Claude Kirchner[1], and Colin Riba[2]

[1] INRIA & LORIA*
[2] INPL & LORIA

Abstract. The confluence of untyped λ-calculus with *unconditional* rewriting has already been studied in various directions. In this paper, we investigate the confluence of λ-calculus with *conditional* rewriting and provide general results in two directions. First, when conditional rules are algebraic. This extends results of Müller and Dougherty for unconditional rewriting. Two cases are considered, whether beta-reduction is allowed or not in the evaluation of conditions. Moreover, Dougherty's result is improved from the assumption of strongly normalizing β-reduction to weakly normalizing β-reduction. We also provide examples showing that outside these conditions, modularity of confluence is difficult to achieve. Second, we go beyond the algebraic framework and get new confluence results using a restricted notion of orthogonality that takes advantage of the conditional part of rewrite rules.

1 Introduction

Rewriting [10] and λ-calculus [3] are two universal computation models which are both used, with their own advantages, in programming language design and implementation, as well as for the foundation of logical frameworks and proof assistants. Among other things, λ-calculus allows to manipulate abstractions and higher-order variables, while rewriting is traditionally well suited for defining functions over data-types and for dealing with equality.

Starting from Klop's work on higher-order rewriting and because of their complementarity, many frameworks have been designed with a view to integrate these two formalisms. This integration has been handled either by enriching first-order rewriting with higher-order capabilities, by adding to λ-calculus algebraic features or, more recently, by a uniform integration of both paradigms. In the first case, we find the works on combinatory reduction systems [17] and other higher-order rewriting systems [20] each of them subsumed by van Oostrom and van Raamsdonk's axiomatization of HORS [23]. The second case concerns the more atomic combination of λ-calculus with term rewriting [15, 5] and the last category the rewriting calculus [9, 4].

Despite this strong interest in the combination of both concepts, few works have considered *conditional* higher-order rewriting in λ-calculus. This is of

* UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP, Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex, France.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 382–397, 2006.
© Springer-Verlag Berlin Heidelberg 2006

particular interest for both computation and deduction. Indeed, conditional rewriting appears to be very convenient when programming with rewrite rules and its combination with higher-order features provides a quite agile background for the combination of algebraic and functional programming. This is also of main use in proof assistants based on the de Bruijn-Curry-Howard isomorphism where, as emphasized in *deduction modulo* [13, 5], rewriting capabilities for defining functions and proving equalities automatically is clearly of great interest when making large proof developments. Furthermore, while many confluence proofs often rely on termination and local confluence, in some cases, confluence may be necessary for proving termination (*e.g.* with type-level rewriting or strong elimination [5]). It is therefore of crucial interest to have also criteria for the preservation of confluence when combining conditional rewriting and β-reduction without assuming the termination of the combined relation. In particular, assuming the termination of just one of the two relations is already of interest.

The earliest work on preservation of confluence when combining typed λ-calculus and first-order rewriting concerns the simple type discipline [7] and the result has been extended to polymorphic λ-calculus in [8]. Concerning untyped λ-calculus, the result was shown in [19] for left-linear rewriting. It is extended as a modularity result for higher order rewriting in [23]. In [12], it is shown that left-linearity is not necessary provided that terms considered are strongly β-normalizable and are well-formed with respect to the declared arity of symbols, a property that we call here *arity-compliance.* Higher-order conditional rewriting is studied in [1] and the confluence result relies on joinability of critical pairs, hence on termination of the combined rewrite relation. Another form of higher-order conditional rewriting is considered in [22]. It concerns confluence results for a very general form of orthogonal systems. These systems are related to those presented in Sect. 5. If modularity properties have been investigated in the pure first-order conditional case (*e.g.* [18, 14]), to the best of our knowledge, there was up to now no result on *preservation* of confluence when β-reduction is added to *conditional* rewriting.

In this paper, we study the confluence property of the combination of β-reduction with a confluent conditional rewrite system. This of course should rely on a clear understanding of the conditional rewrite relation under use and, as usual, the ways the matching is performed and instantiated conditions are decided are crucial.

So, we start from λ-terms with curried constants and among them we distinguish *applicative* terms that contain no abstraction and *algebraic* terms that furthermore have no active variables, *i.e.* variables occurring in the left-hand side of an application. In this paper, we always consider algebraic left-hand sides. So, rewriting does not use higher-order pattern-matching but just syntactic matching. Furthermore, we consider two rewrite relations induced by a set of conditional rules. $\rightarrow_{\mathcal{A}}$ is the conditional rewrite relation where the conditions are checked *without* considering β-reduction and $\rightarrow_{\mathcal{B}}$ is the conditional rewrite relation where β-reduction is allowed when evaluating the conditions. Then,

we study the confluence of the relations $\rightarrow_{\beta\cup\mathcal{A}}$ and $\rightarrow_{\beta\cup\mathcal{B}}$, the respective combinations of $\rightarrow_{\mathcal{A}}$ and $\rightarrow_{\mathcal{B}}$ with β-reduction. This is made precise in Sect. 2 and accompanied of relevant examples.

We know that adding β-reduction to a confluent non left-linear algebraic rewriting system results in a non confluent relation. Of course, with conditional rewriting, non-linearity can be simulated by linear systems. Extending the result of Müller [19], we prove in Sect. 3 that confluence of $\rightarrow_{\beta\cup\mathcal{A}}$ follows from confluence of $\rightarrow_{\mathcal{A}}$ when conditional rules are applicative, left-linear and do not allow their condition to test for equality of open terms. Such rules are called *semi-closed*. We also adapt to conditional rewriting the method of Dougherty [12] and extend it to show that for a large set of *weakly* β-normalizing terms, the left-linearity and semi-closed hypotheses can be dropped provided the rules are algebraic and terms are arity-compliant.

We then turn in Sect. 4 to the confluence modularity of $\rightarrow_{\beta\cup\mathcal{B}}$ for rules with algebraic right-hand side. In this case, we show that arity-compliance is a sufficient condition to deduce confluence of $\rightarrow_{\beta\cup\mathcal{B}}$ from confluence of $\rightarrow_{\beta\cup\mathcal{A}}$ (hence of $\rightarrow_{\mathcal{A}}$). This is done first for left-linear semi-closed systems, a restriction that we also show to be superfluous when considering only *weakly* β-normalizing terms.

The case of non-algebraic rules is handled in Sect. 5. Such rules can contain active variables and abstractions in right-hand sides or in conditions (but still not in left-hand sides). In this case, the confluence of $\rightarrow_{\beta\cup\mathcal{B}}$ no more follows from the confluence of $\rightarrow_{\mathcal{A}}$ nor of $\rightarrow_{\beta\cup\mathcal{A}}$. We show that the confluence of $\rightarrow_{\beta\cup\mathcal{B}}$ holds under a syntactic condition, called *orthonormality* ensuring that if two rules overlap at a non-variable position, then their conditions cannot be both satisfied. An orthonormal system is therefore orthogonal, and the confluence of $\rightarrow_{\mathcal{B}\cup\beta}$ follows using usual proof methods.

We assume some familiarity with λ-calculus [3] and conditional rewriting [11, 21] but we recall the main notations in the next section. By lack of place, the main proofs are only sketched here. They are detailed in [6].

2 General Definitions

This section introduces the main notions of the paper. We use λ-terms with curried constants.

Definition 1 (Terms). *We assume given a set $\mathcal{F}$ of* function symbols *and an infinite set $\mathcal{X}$ of variables. The set $\mathcal{T}$ of* terms *is inductively defined as follows:*

$$t, u \in \mathcal{T} ::= f \in \mathcal{F} \mid x \in \mathcal{X} \mid tu \mid \lambda x.t$$

A term is applicative *if it contains no abstraction and* algebraic *("not variable-applying" in [19]) if it furthermore contains no subterm of the form xt with $x \in \mathcal{X}$. We use $\boldsymbol{t}$ to denote a sequence of terms $t_1, \dots, t_n$ of length $|\boldsymbol{t}| = n$.*

As usual, terms are considered modulo α-conversion. Let $\mathrm{FV}(t)$ be the set of variables free in t. We denote by $t\sigma$ the capture-avoiding application of the

substitution σ to the term t. By $\{\boldsymbol{x} \mapsto \boldsymbol{t}\}$, we denote the substitution σ such that $x_i\sigma = t_i$. As usual, positions in a term are strings over $\{1,2\}$. The subterm of t at position p is denoted by $t|_p$. If t is applicative, the replacement of $t|_p$ by some term u is denoted by $t[u]_p$. A *context* is a term with exactly one free occurrence of a distinguished variable $[]$. If C is an applicative context then $C[t]$ stands for $C[t]_p$, where p is the position of $[]$ in C.

A rewrite relation is a binary relation on terms $\rightarrow$ which is closed by term formation rules : if $s \rightarrow t$ then $\lambda x.s \rightarrow \lambda x.t$, $su \rightarrow tu$ and $us \rightarrow ut$; and by substitution : $s \rightarrow t$ implies $s\sigma \rightarrow t\sigma$. Its inverse is denoted by $\leftarrow$; its reflexive closure by $\rightarrow^=$; its reflexive and transitive closure by $\rightarrow^*$; and its reflexive, symmetric and transitive closure by $\leftrightarrow^*$. The *joinability* relation is $\downarrow = \rightarrow^*\leftarrow^*$. The β-reduction relation is the smallest rewrite relation $\rightarrow_\beta$ such that $(\lambda x.s)t \rightarrow_\beta s\{x \mapsto t\}$. A term t $\rightarrow$*-rewrites* (or $\rightarrow$*-reduces*) to u if $t \rightarrow^* u$ (we omit $\rightarrow$ when clear from the context). We write $\rightarrow_{R\cup S}$ for the union of the relations $\rightarrow_R$ and $\rightarrow_S$. We call *parallel rewrite relation* any reflexive rewrite relation $\rhd$ closed by *parallel application* : $[s \rhd s' \,\&\, t \rhd t'] \Rightarrow st \rhd s't'$.

We now introduce conditional rewriting. Let us emphasize that we consider first-order syntactical matching.

Definition 2 (Conditional rewriting). *A* conditional rewrite system $\mathcal{R}$ *is a set of conditional rewrite rules*[1]*:*

$$d_1 = c_1 \wedge \cdots \wedge d_n = c_n \supset l \rightarrow r$$

where l is a non-variable algebraic term, d_i, c_i and r are arbitrary terms and $\mathrm{FV}(d_i, c_i, r) \subseteq \mathrm{FV}(l)$*. A system is* right-applicative *(resp.* right-algebraic*) if all its right-hand sides are applicative (resp. algebraic). A system is* applicative *(resp.* algebraic*) if all its rules are made of applicative (resp. algebraic) terms.*

The join *rewrite relation induced by $\mathcal{R}$ is usually defined as $\rightarrow_\mathcal{A} = \bigcup_{i\geq 0} \rightarrow_{\mathcal{A}_i}$ [21] where $\rightarrow_{\mathcal{A}_0} = \emptyset$ and for all $i \geq 0$, $\rightarrow_{\mathcal{A}_{i+1}}$ is the smallest rewrite relation such that for all rule $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r \in \mathcal{R}$, for all substitution σ, if $\boldsymbol{d}\sigma \downarrow_{\mathcal{A}_i} \boldsymbol{c}\sigma$ then $l\sigma \rightarrow_{\mathcal{A}_{i+1}} r\sigma$. This relation is sometimes called the* standard conditional rewrite relation.

*We define the β-*standard *rewrite relation induced by $\mathcal{R}$ as $\rightarrow_\mathcal{B} = \bigcup_{i\geq 0} \rightarrow_{\mathcal{B}_i}$ where $\rightarrow_{\mathcal{B}_0} = \emptyset$ and for all $i \geq 0$, $\rightarrow_{\mathcal{B}_{i+1}}$ is the smallest rewrite relation such that for all rule $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r \in \mathcal{R}$, for all σ, if $\boldsymbol{d}\sigma \downarrow_{\mathcal{B}_i\cup\beta} \boldsymbol{c}\sigma$ then $l\sigma \rightarrow_{\mathcal{B}_{i+1}} r\sigma$.*

If $\rightarrow_{\mathcal{A}_i}$ is confluent for all $i \geq 0$, we say that $\rightarrow_\mathcal{A}$ is level confluent*. It is* shallow confluent *when $\rightarrow^*_{\mathcal{A}_i}$ and $\rightarrow^*_{\mathcal{A}_j}$ commute for all $i, j \geq 0$.*

Other forms of conditional rewriting appear in the literature [11]. *Natural* rewriting is obtained by taking $\leftrightarrow^*_\mathcal{A}$ instead of $\downarrow_\mathcal{A}$ in the evaluation of conditions. *Oriented* rewriting is obtained by taking $\rightarrow^*_\mathcal{A}$. A particular case of both standard and oriented rewriting is *normal* rewriting, in which the terms $\boldsymbol{c}$ are closed and in $\rightarrow_\mathcal{A}$-normal form.

[1] The symbol $=$ does not need to be interpreted by a symmetric relation.

Examples. We begin by some basic functions on lists.

$$\begin{array}{lll} \mathsf{car}\ (x :: l) \to x & \mathsf{cdr}\ (x :: l) \to l & \mathsf{get}\ l\ 0 \to \mathsf{car}\ l \\ \mathsf{car}\ [\,] \to \mathsf{err} & \mathsf{cdr}\ [\,] \to \mathsf{err} & \mathsf{get}\ l\ (\mathsf{s}\ n) \to \mathsf{get}\ (\mathsf{cdr}\ l)\ n \end{array}$$

$$\begin{array}{ll} \mathsf{len}\ [\,] \to 0 & \mathsf{filter}\ p\ [\,] \to [\,] \\ \mathsf{len}\ (x :: l) \to \mathsf{s}\ (\mathsf{len}\ l) & p\ x = \mathsf{tt} \supset \mathsf{filter}\ p\ (x :: l) \to x :: (\mathsf{filter}\ p\ l) \\ & p\ x = \mathsf{ff} \supset \mathsf{filter}\ p\ (x :: l) \to \mathsf{filter}\ p\ l \end{array}$$

Define $>$ with $>\ (\mathsf{s}\,x)\ 0\ \to\ \mathsf{tt}$, $>\ 0\ y\ \to\ \mathsf{ff}$ and $>\ (\mathsf{s}\,x)\ (\mathsf{s}\,y)\ \to\ >\ x\ y$. We can now define app such that $\mathsf{app}\ f\ n\ l$ applies f to the nth element of l. It uses ap as an auxiliary function:

$$\begin{array}{ll} >\ (\mathsf{len}\ l)\ n = \mathsf{tt} \supset \mathsf{app}\ f\ n\ l \to \mathsf{ap}\ f\ n\ l & \mathsf{ap}\ f\ 0\ l \to f\ (\mathsf{car}\ l) :: \mathsf{cdr}\ l \\ >\ (\mathsf{len}\ l)\ n = \mathsf{ff} \supset \mathsf{app}\ f\ n\ l \to \mathsf{err} & \mathsf{ap}\ f\ (\mathsf{s}\ n)\ l \to \mathsf{car}\ l :: \mathsf{ap}\ f\ n\ (\mathsf{cdr}\ l) \end{array}$$

We represent first-order terms as trees with nodes $\mathsf{nd}\ y\ l$ where y is intended to be a label and l the list of sons.

Positions are lists of integers and $\mathsf{occ}\ u\ t$ tests if u is an occurrence of t. We define it with $\mathsf{occ}\ [\,]\ t\ \to\ \mathsf{tt}$ and

$$\begin{array}{l} >\ (\mathsf{len}\ l)\ x = \mathsf{ff} \supset \mathsf{occ}\ (x :: o)\ (\mathsf{nd}\ y\ l) \to \mathsf{ff} \\ >\ (\mathsf{len}\ l)\ x = \mathsf{tt} \supset \mathsf{occ}\ (x :: o)\ (\mathsf{nd}\ y\ l) \to \mathsf{occ}\ o\ (\mathsf{get}\ l\ x) \end{array}$$

To finish, $\mathsf{rep}\ t\ o\ s$ replaces by s the subterm of t at occurrence o. Its rules are $\mathsf{occ}\ u\ t = \mathsf{tt} \supset \mathsf{rep}\ t\ o\ s \to \mathsf{re}\ t\ o\ s$ and $\mathsf{occ}\ u\ t = \mathsf{ff} \supset \mathsf{rep}\ t\ o\ s \to \mathsf{err}$. The rules $\mathsf{re}\ s\ [\,]\ t\ \to\ s$ and $\mathsf{re}\ (\mathsf{nd}\ y\ l)\ (x :: o)\ s\ \to\ \mathsf{nd}\ y\ (\mathsf{app}\ (\lambda z.\mathsf{re}\ z\ o\ s)\ x\ l)$ define the function re.

The system Tree that consists of rules defining car cdr, get, len and occ is algebraic. Rules of app and ap are right-applicatives and those for filter contain in their conditions the variable p in active position. *This* definition of re involves a λ-abstraction in a right hand side. In Sect. 5, we prove confluence of the relation $\to_{\beta\cup\mathcal{B}}$ induced by the whole system.

3 Confluence of $\to_\beta$ with Conditional Rewriting

In this section, we study the confluence of $\to_{\beta\cup\mathcal{A}}$. The simplest result is the preservation of confluence when $\mathcal{R}$ can not check arbitrary equalities (Sect. 3.1). In Sect. 3.2, we consider more general systems and prove that the confluence of $\to_{\beta\cup\mathcal{A}}$ follows from the confluence of $\to_\mathcal{A}$ on terms having a β-normal form of a peculiar kind.

In [19], Müller shows that the union of β-reduction and the rewrite relation $\to_\mathcal{A}$ induced by a left-linear non-conditional applicative system is confluent as soon as $\to_\mathcal{A}$ is. This result is generalized as modularity result for higher-order rewriting in [23].

The importance of left-linearity is known since Klop [16]. We exemplify it with Breazu-Tannen's counter-example [7]. The rules $-\ x\ x\ \to\ 0$ and $-\ (\mathsf{s}\ x)\ x\ \to\ \mathsf{s}\ 0$

are optimization rules for minus. Together with usual rules defining this function, they induce a confluent rewrite relation. With the fixpoint combinators of the λ-calculus, we can build a term $Y \to_\beta^* \mathsf{s}\, Y$. This term makes the application of the two rules above possible on β-reducts of $-\ Y\ Y$, leading to an unjoinable peak : $0 \leftarrow_\mathcal{A} -\ Y\ Y \to_\beta^* -\ (\mathsf{s}\, Y)\ Y \to_\mathcal{A} \mathsf{s}\, 0$.

With conditional rewriting, we do not need non-linear matching to distinguish $-\ (\mathsf{s}\ x)\ x$ from $-\ x\ x$, since this can be done within the conditions. The previous system can be encoded into a left-linear conditional system with the rules $x = y \supset -\, x\, y \to 0$ and $\mathsf{s}\, x = y \supset -\, x\, y \to \mathsf{s}\, 0$. Of course, the relation $\to_\mathcal{A}$ is still confluent. However, the same unjoinable peak starting from $-\ Y\ Y$ makes fail the confluence of $\to_{\beta\cup\mathcal{A}}$.

There are two ways to overcome the problem: limiting the power of rewriting or limiting the power of β-reduction. The first way is treated in Sect. 3.1, in which we limit the comparison power of conditional rewriting by restricting ourselves to left-linear and *semi-closed* systems. This can also be seen as a way, from the point of view of rewriting, to isolate the effect of fixpoints: since two distinct occurrences of Y can not be compared, they can be unfolded independently from each other.

Then, in Sect. 3.2, we limit the power of $\to_\beta$ by restricting ourselves to sets of terms having a special kind of β-normal-form. This amounts to only consider terms in which fixpoints do not have the ability to modify the result of $\to_{\beta\cup\mathcal{A}}$. In fact, it is sufficient that they do not modify the result of $\to_\beta$ alone. More precisely, fixpoints are allowed when they are eliminated by head β-reductions.

3.1 Confluence of Left-Linear Semi-closed Systems

We now introduce semi-closed systems.

Definition 3 (Semi-closed systems). *A system is* semi-closed *if in every rule* $\boldsymbol{d} = \boldsymbol{c} \supset l \to r$*, each* c_i *is algebraic and closed.*

The system Tree of Sect. 2 is left-linear and semi-closed. Given a semi-closed left-linear system, we show that confluence of $\to_{\beta\cup\mathcal{A}}$ follows from confluence of $\to_\mathcal{A}$. This follows from a weak commutation of $\to_\mathcal{A}$ and Tait and Martin-Löf β-parallel reduction relation $\rhd_\beta$, defined as the smallest parallel rewrite relation (Sect. 2) closed by the rule (*beta*) [3]:

$$(beta) \quad \frac{s \rhd_\beta s' \quad t \rhd_\beta t'}{(\lambda x.s)t \rhd_\beta s'\{x \mapsto t'\}}$$

We will use some well known properties of $\rhd_\beta$. If $\sigma \rhd_\beta \sigma'$ then $s\sigma \rhd_\beta s\sigma'$; this is the one-step reduction of parallel redexes. We can also simulate β-reduction: $\to_\beta \subseteq \rhd_\beta \subseteq \to_\beta^*$. And third, $\rhd_\beta$ has the diamond property: $\lhd_\beta \rhd_\beta \subseteq \rhd_\beta \lhd_\beta$. This corresponds to the fact that any complete development of $\to_\beta$ can be done in *one* $\rhd_\beta$-step.

Müller [19] uses a weaker parallelization of $\to_\beta$: its relation is defined w.r.t. the applicative structure of terms only and does not reduces in one step nested

β-redexes. Consequently, it does not enjoy the diamond property on which we rely in Sect. 4. Nested parallelizations (corresponding to complete developments) are already used in [23] for their confluence proof of HORS. However, our method inherits more from [19] than [23], as we use complete developments of $\rightarrow_\beta$ only, whereas complete developments of $\rightarrow_\beta$ *and* of $\rightarrow_\mathcal{A}$ are used for the modularity result of [23].

Proposition 4. *Let $\mathcal{R}$ be a semi-closed, left-linear and right-applicative system and assume that $\rightarrow^*_{\mathcal{A}_{i-1}}$ commutes with $\rightarrow^*_\beta$. For any rule $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r \in \mathcal{R}$ and substitution σ, if $u \lhd_\beta l\sigma \rightarrow_{\mathcal{A}_i} r\sigma$, then there exists σ' such that $u = l\sigma' \rightarrow_{\mathcal{A}_i} r\sigma' \lhd_\beta r\sigma$.*

Proof Sketch. Since l is algebraic and linear, there is a substitution σ' such that $\sigma \rhd_\beta \sigma'$ and $u = l\sigma'$. It follows that $r\sigma \rhd_\beta r\sigma'$ and it remains to show that $\boldsymbol{d}\sigma' \downarrow_{\mathcal{A}_{i-1}} \boldsymbol{c}\sigma'$. Since $l\sigma \rightarrow_{\mathcal{A}_i} r\sigma$, there is $\boldsymbol{v}$ such that $\boldsymbol{d}\sigma \rightarrow^*_{\mathcal{A}_{i-1}} \boldsymbol{v} \leftarrow^*_{\mathcal{A}_{i-1}} \boldsymbol{c}\sigma$. Thus, $\boldsymbol{d}\sigma \rhd^*_\beta \boldsymbol{d}\sigma'$ and, by assumption, there is $\boldsymbol{v}'$ such that $\boldsymbol{d}\sigma' \rightarrow^*_{\mathcal{A}_{i-1}} \boldsymbol{v}' \lhd^*_\beta \boldsymbol{v}$. Since $\boldsymbol{c}$ is algebraic and closed, we have $\boldsymbol{c}\sigma = \boldsymbol{c}$ and $\boldsymbol{v}$ in β-normal form. Hence, $\boldsymbol{v}' = \boldsymbol{v}$ and $\boldsymbol{d}\sigma' \downarrow_{\mathcal{A}_{i-1}} \boldsymbol{c}$. □

Lemma 5 (Commutation of $\rightarrow_\mathcal{A}$ and $\rhd_\beta$). *If $\mathcal{R}$ is a semi-closed left-linear right-applicative system, then $\lhd^*_\beta \rightarrow^*_\mathcal{A} \subseteq \rightarrow^*_\mathcal{A} \lhd^*_\beta$.*

Proof Sketch. The result follows from the commutation of $\rightarrow^*_{\mathcal{A}_i}$ and $\rhd^*_\beta$ for all $i \geq 0$. The case $i = 0$ is trivial. For $i > 0$, there are three steps. First, we show by induction on the definition of the parallel rewrite relation $\rhd_\beta$ that if $u \lhd_\beta s \rightarrow_{\mathcal{A}_i} t$ then there exists v such that $u \rightarrow^*_{\mathcal{A}_i} v \lhd_\beta t$. If u is s this is obvious. If s is an abstraction, the result follows from induction hypothesis (IH) and the context closure of $\rightarrow_{\mathcal{A}_i}$ (CC). If $s = s_1 s_2$, there are two cases: if $t = t_1 t_2$ with $s_k \rightarrow^=_{\mathcal{A}_i} t_k$ then we conclude by (IH) and (CC). Otherwise, we use Prop. 4.

Second, use induction on the number of $\mathcal{A}_i$-steps to show that $\lhd_\beta \rightarrow^*_{\mathcal{A}_i} \subseteq \rightarrow^*_{\mathcal{A}_i} \rhd_\beta$. Finally, to conclude that $\lhd^*_\beta \rightarrow^*_{\mathcal{A}_i} \subseteq \rightarrow^*_{\mathcal{A}_i} \lhd^*_\beta$, use an induction on the number of $\rhd_\beta$-steps. □

A direct application of Hindley-Rosen's Lemma offers then the preservation of confluence.

Theorem 6 (Confluence of $\rightarrow_{\beta\cup\mathcal{A}}$). *Let $\mathcal{R}$ be a semi-closed left-linear right-applicative system. If $\rightarrow_\mathcal{A}$ is confluent then so is $\rightarrow_{\beta\cup\mathcal{A}}$.*

For the system Tree of Sect. 2, the relation $\rightarrow_\mathcal{A}$ is confluent. As the rules are left-linear and semi-closed, Theorem 6 applies and $\rightarrow_{\beta\cup\mathcal{A}}$ is confluent.

3.2 Confluence on Weakly β-Normalizing Terms

We now turn to the problem of dropping the left-linearity and semi-closure conditions.

As seen above, fixpoint combinators make the commutation of $\rightarrow^*_\beta$ and $\rightarrow^*_\mathcal{A}$ fail when rewriting involves equality tests between open terms. When using weakly

β-normalizing terms, we can project rewriting on β-normal forms (βnf), thus eliminating fixpoints as soon as they are not significant for the reduction.

Hence, we seek to obtain $\beta nf(s) \rightarrow^*_{\mathcal{A}} \beta nf(t)$ whenever $s \rightarrow^*_{\beta\cup\mathcal{A}} t$. This requires three important properties.

First, β-normal forms should be stable by rewriting. Hence, we assume that right-hand sides are algebraic. Moreover, we re-introduce some information from the algebraic framework, giving maximal arities to function symbols in $\mathcal{F}$.

Second, we need normalizing β-derivations to commute with rewriting. This follows from using the leftmost-outermost strategy of λ-calculus [3].

Finally, we need rule conditions to be algebraic. Indeed, consider the rule $x\,\mathsf{b} = y \supset \mathsf{f}\,x\,y \rightarrow \mathsf{a}$ that contains an non-algebraic condition. The relation $\rightarrow_{\mathcal{A}}$ is confluent but $\mathsf{a} \leftarrow^*_{\beta\cup\mathcal{A}} \mathsf{f}\,(\lambda x.x)\,((\lambda z.z)(\lambda x.x)\,\mathsf{b}) \rightarrow^*_{\beta} \mathsf{f}\,(\lambda x.x)\,\mathsf{b}$ is an unjoinable critical peak.

Definition 7 (Arity-compliance). *We assume that every symbol $f \in \mathcal{F}$ is equipped with an arity $\alpha_f \geq 0$. A term is* arity-compliant *if it contains no subterm of the form $f\boldsymbol{t}$ with $f \in \mathcal{F}$ and $|\boldsymbol{t}| > \alpha_f$. A rule $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r$ is* almost arity-compliant *if l and r are arity-compliant and l is of the form $f\boldsymbol{l}$ with $|\boldsymbol{l}| = \alpha_f$. A rule is* arity-compliant *if, furthermore, $\boldsymbol{d}$ and $\boldsymbol{c}$ are arity-compliant. Let $\mathcal{U}$ be the set of terms having an arity-compliant β-normal form.*

Remark that a higher-order rule (with active variables and abstractions) can be arity-compliant.

Arity-compliance is useful because it prevents collapsing rules from creating β-redexes. For example, the rule $\mathsf{id}\ x \rightarrow x$ forces the arity of id to be 1. Hence the term $\mathsf{id}\,(\lambda x.x)\,y$ is not arity-compliant. Moreover it is a β-normal form that $\rightarrow_{\mathcal{A}}$-reduces to the β-redex $(\lambda x.x)y$. It is then easy to build an arity-uncompliant term that makes the preservation of confluence to fail. Let $Y = \omega_{\mathsf{s}}\omega_{\mathsf{s}}$ with $\omega_{\mathsf{s}} = \lambda x.\mathsf{s}\,x\,x$. The term $-\,(\mathsf{id}\,\omega_{\mathsf{s}}\omega_{\mathsf{s}})\,(\mathsf{id}\,\omega_{\mathsf{s}}\omega_{\mathsf{s}})$ is an arity-uncompliant β-normal form. Reducing the id's leads to $-\,Y\,Y$ which is the head of an unjoinable critical peak.

However, we do not assume that every term at hand is arity-compliant. Indeed, a term that has an arity-compliant β-normal form does not need to be arity-compliant itself. More precisely, for a weakly β-normalizing term, the leftmost-outermost strategy (for $\rightarrow_\beta$) never evaluates subterms that are not β-normalizing and it follows that such subterms may be arity-uncompliant without disturbing the projection on β-normal forms.

The point is the well-foundedness of the leftmost-outermost strategy for $\rightarrow_\beta$ on weakly β-normalizing terms [3]. This strategy can be described by means of *head* β-reductions, that are easily shown to commute with (parallel) conditional rewriting. Any λ-term can be written $\lambda\boldsymbol{x}.v\,a_0\,a_1\ldots a_n$ where either $v \in \mathcal{X}\cup\mathcal{F}$ (a) or v is a λ-abstraction (b). We denote by $\rightarrow_h$ the head β-step $\lambda\boldsymbol{x}.(\lambda y.b)a_0\boldsymbol{a} \rightarrow_h \lambda\boldsymbol{x}.b\{y \mapsto a_0\}\boldsymbol{a}$. Let $s \succ t$ iff either s is of the form (b) and $s \rightarrow_h t$, or s is of the form (a) with $n \geq 1$ and $t = a_i$ for some $i \geq 0$. In the latter case, the free variables of t can be bound in s. Hence, t can be a subterm of a term α-equivalent to s ; for instance $\lambda x.\mathsf{f}x \succ y$ for all $y \in \mathcal{X}$.

Lemma 8. *Let $\mathcal{WN}$ be the set of weakly β-normalizing terms ; (i) if $s \in \mathcal{WN}$ and $s \succ t$ then $t \in \mathcal{WN}$, (ii) $\succ$ is well-founded on $\mathcal{WN}$.*

It follows that we can reason by well-founded induction on $\succ$. For all $i \geq 0$, we use a nested parallelization of $\rightarrow_{\mathcal{A}_i}$. It corresponds to the one used in [23], that can be seen as a generalization of Tait and Martin-Löf parallel relation. As for $\rhd_\beta$ and $\rightarrow_\beta$, in the orthogonal case, a complete development of $\rightarrow_{\mathcal{A}_i}$ can be simulated by *one step* $\rhd_{\mathcal{A}_i}$-reduction. This relation is also an adaptation to conditional rewriting of the parallelization used in [12].

Definition 9 (Conditional nested parallel relations). *For all $i \geq 0$, let $\rhd_{\mathcal{A}_i}$ be the smallest parallel rewrite relation closed by:*

$$(rule) \quad \frac{\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r \in \mathcal{R} \quad l\sigma \rightarrow_{\mathcal{A}_i} r\sigma \quad \sigma \rhd_{\mathcal{A}_i} \theta}{l\sigma \rhd_{\mathcal{A}_i} r\theta}$$

Recall that $l\sigma \rightarrow_{\mathcal{A}_i} r\sigma$ is ensured by $\boldsymbol{d}\sigma \downarrow_{\mathcal{A}_{i-1}} \boldsymbol{c}\sigma$. These relations enjoy some nice properties: (1) $\rightarrow_{\mathcal{A}_i} \subseteq \rhd_{\mathcal{A}_i} \subseteq \rightarrow^*_{\mathcal{A}_i}$, (2) $s \rhd_{\mathcal{A}_i} t \Rightarrow u\{x \mapsto s\} \rhd_{\mathcal{A}_i} u\{x \mapsto t\}$ and (3) $[s \rhd_{\mathcal{A}_i} t \;\&\; u \rhd_{\mathcal{A}_i} v] \;\Rightarrow\; u\{x \mapsto s\} \rhd_{\mathcal{A}_i} v\{x \mapsto t\}$. The last one implies commutation of $\rhd_{\mathcal{A}_i}$ and $\rightarrow_h$. Commutation of rewriting with head β-reduction has already been coined in [2]. We now turn to the main lemma.

Lemma 10. *Let $\mathcal{R}$ be an arity-compliant algebraic system. If $s \in \mathcal{U}$ and $s \rightarrow^*_{\beta\cup\mathcal{A}} t$, then $t \in \mathcal{U}$ and $\beta nf(s) \rightarrow^*_{\mathcal{A}} \beta nf(t)$.*

Proof Sketch. We show by induction on i the property for $\rightarrow^*_{\beta\cup\mathcal{A}_i}$. We denote by (I) the corresponding induction hypothesis. The case $i = 0$ is trivial. Assume that $i > 0$. An induction on the number of $\rightarrow_{\beta\cup\mathcal{A}_i}$-steps leads us to prove that $\beta nf(s) \rhd_{\mathcal{A}_i} \beta nf(t)$ whenever $s \rhd_{\mathcal{A}_i} t$ and s has an arity-compliant β-normal form. We reason by induction on $\succ$.

First (1), assume that s is of the form (a). If no rule is reduced at its head, the result follows from induction hypothesis on $\succ$. Otherwise, there is a rule $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r$ such that $s = \lambda\boldsymbol{x}.l\sigma\boldsymbol{a}$ and $t = \lambda\boldsymbol{x}.r\theta\boldsymbol{b}$ with $l\sigma \rhd_{\mathcal{A}_i} r\theta$ and $\boldsymbol{d}\sigma \downarrow_{\mathcal{A}_{i-1}} \boldsymbol{c}\sigma$. Since l is algebraic, $\beta nf(s)$ is of the form $\lambda\boldsymbol{x}.l\sigma'\boldsymbol{a}'$ where $\sigma' = \beta nf(\sigma)$ and $\boldsymbol{a}' = \beta nf(\boldsymbol{a})$. Since $\beta nf(s)$ is arity-compliant, $\boldsymbol{a}' = \emptyset$, hence $\boldsymbol{a} = \emptyset$ and $s = \lambda\boldsymbol{x}.l\sigma$. Therefore, because $l\sigma \rhd_{\mathcal{A}_i} r\theta$, we have $\boldsymbol{b} = \emptyset$ and $t = \lambda\boldsymbol{x}.r\theta$. It remains to show that t has an arity-compliant normal form and that $\beta nf(s) = \lambda\boldsymbol{x}.l\sigma' \rhd_{\mathcal{A}_i} \beta nf(t)$. Because l is algebraic, its variables are $\prec^+ l$. We can then apply induction hypothesis on $\sigma \rhd_{\mathcal{A}_i} \theta$. It follows that θ has an arity-compliant normal form θ' with $\sigma' \rhd_{\mathcal{A}_i} \theta'$. Since r is algebraic, $\lambda\boldsymbol{x}.r\theta'$ is the (arity-compliant) β-normal form of t. Hence it remains to show that $l\sigma' \rhd_{\mathcal{A}_i} r\theta'$. Because $\sigma' \rhd_{\mathcal{A}_i} \theta'$, it suffices to prove that $l\sigma' \rightarrow_{\mathcal{A}_i} r\sigma'$. Thus, we are done if we show that $\boldsymbol{d}\sigma' \downarrow_{\mathcal{A}_{i-1}} \boldsymbol{c}\sigma'$. Since $\boldsymbol{d}$ and $\boldsymbol{c}$ are algebraic, $\beta nf(\boldsymbol{d}\sigma) = \boldsymbol{d}\sigma'$ and $\beta nf(\boldsymbol{c}\sigma) = \boldsymbol{c}\sigma'$. Now, since $\boldsymbol{d}$ is algebraic and arity-compliant and σ' is arity compliant, $\boldsymbol{d}\sigma'$ is arity-compliant. The same holds for $\boldsymbol{c}\sigma'$. Hence we conclude by applying induction hypothesis (I) on $\boldsymbol{d}\sigma \downarrow_{\mathcal{A}_{i-1}} \boldsymbol{c}\sigma$.

Second (2), when s is of the form (b) we head β-normalize it and obtain a term s' of the form (a) having an arity-compliant β-normal form. Using commutation

of $\rhd_{\mathcal{A}_i}$ and $\rightarrow_h$, we obtain a term t' such that $s' \rhd_{\mathcal{A}_i} t'$. Since $s \succ^+ s'$, we can reason as in case (1). $\square$

The preservation of confluence is a direct consequence of the projection on β-normal forms.

Theorem 11. *Let $\mathcal{R}$ be an arity-compliant algebraic system such that $\rightarrow_{\mathcal{A}}$ is confluent. Then, $\rightarrow_{\beta\cup\mathcal{A}}$ is confluent on $\mathcal{U}$.*

Comparison with Dougherty's work. This section is an extension of [12]. We give a further exploration of the idea that preservation of confluence, when using hypothesis on $\rightarrow_\beta$, should be independent from any typing discipline for the λ-calculus.

Moreover, we extend its result in three ways. First, we adapt it to conditional rewriting. Second, we allow nested symbols in lhs to be applied to less arguments than their arity. And third, we use weakly β-normalizing terms whose normal forms are arity-compliant ; whereas Dougherty uses the set of strongly normalizing arity-compliant terms which is closed by reduction.

4 Using $\rightarrow_\beta$ in the Evaluation of Conditions

The goal of this section is to give conditions on $\mathcal{R}$ to deduce confluence of $\rightarrow_{\beta\cup\mathcal{B}}$ from confluence of $\rightarrow_{\mathcal{A}}$. We achieve this by exhibiting two different criteria ensuring that

$$\rightarrow^*_{\beta\cup\mathcal{B}} \subseteq \rightarrow^*_\beta \rightarrow^*_{\mathcal{A}} \leftarrow^*_\beta \ . \qquad (\star)$$

The first case concerns left-linear and semi-closed systems. This holds only on some sets of terms that, after Dougherty [12], we call *$\mathcal{R}$-stable*, although our definition of stability does not require strong β-normalization (see Sect. 3.2 and Def. 12). This is an extra hypothesis compared to the result of Sect. 3.1. The second case is a direct extension of Lemma 10 to $\rightarrow_{\beta\cup\mathcal{B}}$. In both cases, we assume the rules to be algebraic and arity-compliant. We are then able to obtain confluence of $\rightarrow_{\beta\cup\mathcal{B}}$ since, in each case, our assumptions ensure that the results of Sect. 3 applies, hence that $\rightarrow_{\beta\cup\mathcal{A}}$ is confluent whenever $\rightarrow_{\mathcal{A}}$ is.

It is important to underline the meaning of ($\star$). Given an arity-compliant *algebraic* rule $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r$, every β-redex occurring in $\boldsymbol{d}\sigma$ or $\boldsymbol{c}\sigma$ also occurs in $l\sigma$. Then, ($\star$) means that there is a β-reduction starting from $l\sigma$ that reduces these redexes and produce a substitution σ' such that $l\sigma \rightarrow^*_\beta l\sigma' \rightarrow_{\mathcal{A}} r\sigma' \leftarrow^*_\beta r\sigma$. In other words, if the conditions are satisfied with σ and $\rightarrow_{\beta\cup\mathcal{B}}$ (*i.e.* $\boldsymbol{d}\sigma \downarrow_{\beta\cup\mathcal{B}} \boldsymbol{c}\sigma$), then they are satisfied with σ' and $\rightarrow_{\mathcal{A}}$ (*i.e.* $\boldsymbol{d}\sigma' \downarrow_{\mathcal{A}} \boldsymbol{c}\sigma'$).

We now give some examples of non arity-compliant or non algebraic rules in which, at the same time, ($\star$) fails and $\rightarrow_{\beta\cup\mathcal{B}}$ is not confluent whereas $\rightarrow_{\beta\cup\mathcal{A}}$ for (1), (3), (4) and at least $\rightarrow_{\mathcal{A}}$ for (2) is.

(1)		$\mathsf{g}x \rightarrow x\mathsf{c}$	$\mathsf{g}x = \mathsf{d} \supset \mathsf{f}x \rightarrow \mathsf{a}$	$\mathsf{f}x \rightarrow \mathsf{b}$
(2)			$x\mathsf{c} = \mathsf{d} \supset \mathsf{f}x \rightarrow \mathsf{a}$	$\mathsf{f}x \rightarrow \mathsf{b}$
(3)		$\mathsf{h}x \rightarrow x$	$\mathsf{h}x\mathsf{c} = \mathsf{d} \supset \mathsf{f}x \rightarrow \mathsf{a}$	$\mathsf{f}x \rightarrow \mathsf{b}$
(4)	$\mathsf{h}xy \rightarrow \mathsf{g}xy$	$\mathsf{g}x \rightarrow x$	$\mathsf{h}x\mathsf{c} = \mathsf{d} \supset \mathsf{f}x \rightarrow \mathsf{a}$	$\mathsf{f}x \rightarrow \mathsf{b}$

The first and second examples respectively contain a rule with a non algebraic right-hand side and a rule with a non algebraic condition. Examples (3) and (4) use non arity-compliant terms, in the conditional part and in the right-hand side of a rule respectively. For these four examples, the step $\mathsf{f}(\lambda x.\mathsf{d}) \rightarrow_{\mathcal{B}} \mathsf{a}$ is not in $\rightarrow_\beta^* \rightarrow_\mathcal{A}^* \leftarrow_\beta^*$ and $\mathsf{a} \leftarrow_{\mathcal{B}} \mathsf{f}(\lambda x.\mathsf{d}) \rightarrow_{\mathcal{B}} \mathsf{b}$ is an unjoinable peak.

However, $(\star)$ is by no means a necessary condition ensuring that $\rightarrow_{\beta\cup\mathcal{B}}$ is confluent when $\rightarrow_{\beta\cup\mathcal{A}}$ so is. In the above examples, confluence of $\rightarrow_{\beta\cup\mathcal{B}}$ can be recovered when adding appropriate rules, yet not restoring $(\star)$.

As we are interested in deducing the confluence of $\rightarrow_{\beta\cup\mathcal{B}}$ from the confluence of $\rightarrow_\mathcal{A}$, it is more convenient to take in Def. 2 $\rightarrow_{\mathcal{B}} = \bigcup_{i\geq 0} \rightarrow_{\mathcal{B}_i}$ with $\rightarrow_{\mathcal{B}_0} = \rightarrow_\mathcal{A}$ instead of $\rightarrow_{\mathcal{B}_0} = \emptyset$ (this does not change $\rightarrow_\mathcal{B}$ since $\rightarrow_\mathcal{A} \subseteq \rightarrow_\mathcal{B}$).

4.1 Confluence of Left-Linear Systems

In this paragraph, we prove $(\star)$ provided that rules are arity-compliant, algebraic, left-linear and semi-closed. This inclusion is shown on $\mathcal{R}$-*stable* sets of terms.

Definition 12 ($\mathcal{R}$-stable sets). *Let $\mathcal{R}$ be a set of rules. A set $\mathcal{S}$ is* almost $\mathcal{R}$-stable *if it contains only arity-compliant terms, is stable by subterm and β-reduction, and $C[r\sigma] \in \mathcal{S}$ whenever $C[l\sigma] \in \mathcal{S}$ and $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r \in \mathcal{R}$. An almost $\mathcal{R}$-stable set $\mathcal{S}$ is $\mathcal{R}$-stable if $\boldsymbol{d}\sigma, \boldsymbol{c}\sigma \in \mathcal{S}$ whenever $C[l\sigma] \in \mathcal{S}$ and $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r \in \mathcal{R}$.*

This includes the set of strongly $\rightarrow_{\beta\cup\mathcal{A}}$-normalizable arity-compliant terms and any of its subset closed by subterm and reduction, by using a simple type discipline for instance.

The inclusion $(\star)$ is proved by induction on the stratification of $\rightarrow_\mathcal{B}$ with $\rightarrow_{\mathcal{B}_0} = \rightarrow_\mathcal{A}$. The base case corresponds to $\rightarrow_{\beta\cup\mathcal{A}}^* \subseteq \rightarrow_\beta^* \rightarrow_\mathcal{A}^* \leftarrow_\beta^*$, which does not require rule conditions to be algebraic nor arity-compliant.

The previous examples show however that this may fail in presence of arity-uncompliant or non-algebraic right-hand sides. Note that the result is proved only on almost $\mathcal{R}$-stable sets of terms. Note also that a set containing a term reducible by the first rule of example (4) above is obviously not stable. Finally, note that the β-expansion steps are needed because rules can be duplicating.

Lemma 13. *Let $\mathcal{R}$ be a semi-closed left-linear right-algebraic system. On any almost $\mathcal{R}$-stable set of terms, $\rightarrow_{\beta\cup\mathcal{A}}^* \subseteq \rightarrow_\beta^* \rightarrow_\mathcal{A}^* \leftarrow_\beta^*$.*

Proof Sketch. The proof is in four steps. We begin (1) to show that $\rightarrow_\mathcal{A} \rhd_\beta \subseteq \rhd_\beta \rightarrow_\mathcal{A}^* \lhd_\beta$, reasoning by cases on the step $\rhd_\beta$. This inclusion relies on an important fact of algebraic terms: if s is algebraic and $s\sigma \rhd_\beta v$ then $v \rhd_\beta s\sigma'$ with $\sigma \rhd_\beta^* \sigma'$. From (1), it follows that (2) $\rightarrow_\mathcal{A}^* \rhd_\beta \subseteq \rhd_\beta \rightarrow_\mathcal{A}^* \lhd_\beta^*$, by induction on the number of $\rightarrow_\mathcal{A}$-steps. Then (3), we obtain $\rightarrow_\mathcal{A}^* \rhd_\beta^* \subseteq \rhd_\beta^* \rightarrow_\mathcal{A}^* \lhd_\beta^*$ using an induction on the number of $\rhd_\beta$-steps and the diamond property of $\rhd_\beta$. Finally (4), we deduce that $(\rhd_\beta \cup \rightarrow_\mathcal{A})^* \subseteq \rhd_\beta^* \rightarrow_\mathcal{A}^* \lhd_\beta^*$ by induction on the length of $(\rhd_\beta \cup \rightarrow_\mathcal{A})^*$. □

We now turn to the main result of this subsection. As seen in the previous examples, rules have to be algebraic and arity-compliant.

Lemma 14. *Let $\mathcal{R}$ be a semi-closed left-linear algebraic system. On any $\mathcal{R}$-stable set of terms,* $\to^*_{\beta\cup\mathcal{B}} \subseteq \to^*_\beta \to^*_\mathcal{A} \leftarrow^*_\beta$.

Proof Sketch. The first point is to see that (1) $\to^*_{\mathcal{B}_1} \subseteq \to^*_\beta \to^*_\mathcal{A} \leftarrow^*_\beta$. This is done by induction on the number of $\mathcal{B}_1$-steps, using Lemma 13. We then deduce (2) $\to^*_{\beta\cup\mathcal{B}_1} \subseteq \to^*_\beta \to^*_\mathcal{A} \leftarrow^*_\beta$, by induction on the number of $\to_{\beta\cup\mathcal{B}_1}$-steps. The result follows from an induction on i showing that $\to_{\mathcal{B}_i} \subseteq \to_{\mathcal{B}_1}$. □

Theorem 15. *Assume that $\mathcal{R}$ is a semi-closed left-linear algebraic system. If $\to_\mathcal{A}$ is confluent, then $\to_{\beta\cup\mathcal{B}}$ is confluent on any $\mathcal{R}$-stable set of terms.*

Recall that in this case $\to_{\beta\cup\mathcal{A}}$-confluence follows from $\to_\mathcal{A}$-confluence by Thm. 6.

4.2 Confluence on Weakly β-Normalizing Terms

This subsection concerns the straightforward extension to $\to_\mathcal{B}$ of the results of Sect. 3.2. The definition of $\rhd_{\mathcal{B}_i}$ follows the same scheme as the one of $\rhd_{\mathcal{A}_i}$; the only difference is that $\mathcal{B}_i$ is used everywhere in place of $\mathcal{A}_i$. It follows that given a rule $\boldsymbol{d} = \boldsymbol{c} \supset l \to r$, to have $l\sigma \rhd_{\mathcal{B}_i} r\theta$, we must have $\sigma \rhd_{\mathcal{B}_i} \theta$ and $\boldsymbol{d}\sigma \downarrow_{\beta\cup\mathcal{B}_{i-1}} \boldsymbol{c}\sigma$. The relations $\rhd_{\mathcal{B}_i}$ enjoy the same nice properties as the $\rhd_{\mathcal{A}_i}$'s.

Lemma 16. *Let $\mathcal{R}$ be an arity-compliant algebraic system. If $s \in \mathcal{U}$ and $s \to^*_{\beta\cup\mathcal{B}} t$, then $t \in \mathcal{U}$ and $\beta nf(s) \to^*_\mathcal{A} \beta nf(t)$.*

The only difference in the proof is that the case $i = 0$ is now ensured by Lemma 10 (since $\to_{\mathcal{B}_0} = \to_\mathcal{A}$). The theorem follows easily:

Theorem 17. *Let $\mathcal{R}$ be an arity-compliant algebraic system such that $\to_\mathcal{A}$ is confluent. Then, $\to_{\beta\cup\mathcal{B}}$ is confluent on $\mathcal{U}$.*

5 Orthonormal Systems

In this section, we give a criterion ensuring confluence of $\to_{\beta\cup\mathcal{B}}$ when conditions and right-hand sides possibly contain abstractions and active variables.

This criterion comes from peculiarities of orthogonality with conditional rewriting. In non-conditional rewriting, a system is orthogonal when it is left-linear and has no critical pair. A critical pair comes from the superposition of two different rule left-hand sides at non-variable positions. The general definition of orthogonal conditional systems is the same. But, in conditional rewriting, there can be superpositions of two different rules left-hand sides whose conditions cannot be satisfied with the same substitution. Such critical pairs are said *infeasible* and it could be profitable to consider systems whose critical pairs are all infeasible.

In [21], it is remarked that results on the confluence of natural and normal orthogonal conditional systems should be extended to systems that have no feasible critical pair. But the results obtained this way are not directly applicable since proving unfeasibility of critical pairs may require confluence. In Takahashi's

work [22], conditions can be any predicate P on terms. Confluence is proved with the assumption that they are stable by reduction: if $P\sigma$ holds and $\sigma \rightarrow \theta$, then $P\theta$ holds. For the systems studied in this section, stability of conditions by reduction precisely follows from confluence. Hence the results of [22] do not directly apply.

The purpose of this section is to give a *syntactic* condition on rules that imply unfeasibility of critical pairs, hence confluence.

Definition 18 (Conditional critical pairs). *Given two rules $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r$ and $\boldsymbol{d}' = \boldsymbol{c}' \supset l' \rightarrow r'$, if p is a non-variable position in l and σ is a most general unifier of $l|_p$ and l', then*

$$\boldsymbol{d}\sigma = \boldsymbol{c}\sigma \wedge \boldsymbol{d}'\sigma = \boldsymbol{c}'\sigma \supset (l[r']_p\sigma, r\sigma)$$

is a conditional critical pair*. A critical pair $\boldsymbol{d} = \boldsymbol{c} \supset (s,t)$ is* feasible *for $\rightarrow_{\mathcal{A}}$ (resp. $\rightarrow_{\mathcal{B}}$) if there is a substitution σ such that $\boldsymbol{d}\sigma \downarrow_{\mathcal{A}} \boldsymbol{c}\sigma$ (resp. $\boldsymbol{d}\sigma \downarrow_{\beta\cup\mathcal{B}} \boldsymbol{c}\sigma$).*

As an example, consider the rules used to define occ in Sect. 2. There is a superposition between the left-hand sides of the last two rules giving the critical peak $\mathsf{ff} \leftarrow \mathsf{occ}\,(x :: o)\,(\mathsf{nd}\,y\,l) \rightarrow \mathsf{occ}\,o\,(\mathsf{get}\,l\,x)$. But a peak of this form can occur only if there are two terms s, t such that $\mathsf{tt} \;\leftarrow^* \geq\; (\mathsf{len}\,s)\,t \;\rightarrow^*\; \mathsf{ff}$. Using the stratification of $\rightarrow_{\mathcal{A}}$, the confluence of $\rightarrow_{\mathcal{A}_i}$ implies that this pair is not feasible. Hence the above peak cannot occur with $\rightarrow_{\mathcal{A}_{i+1}}$ and this relation is confluent.

This method can be used on systems with higher-order terms in right-hand sides and conditions, as for example the rules defining app and filter. Hence, it is useful for proving the confluence of $\rightarrow_{\beta\cup\mathcal{B}}$ for systems where this relation does not need to be included in $\leftrightarrow^*_{\beta\cup\mathcal{A}}$. In this section, we generalize the method and apply it on a class of systems called *orthonormal*. As in the previous section, we use stratification of $\rightarrow_{\mathcal{B}}$, but now with $\rightarrow_{\mathcal{B}_0} = \emptyset$. A symbol $f \in \mathcal{F}$ is *defined* if it is the head of a rule left-hand side.

Definition 19 (Orthonormal systems). *A system is* orthonormal *if (1) it is left-linear; (2) in every rule $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r$, the terms in $\boldsymbol{c}$ are closed β-normal forms not containing defined symbols; and (3) for every critical pair $\boldsymbol{d} = \boldsymbol{c} \supset (s,t)$, there exists $i \neq j$ such that $d_i = d_j$ and $c_i \neq c_j$.*

Note that an orthonormal system is left-linear and semi-closed, but does not need to be arity-compliant or algebraic. Note also that the form of the conditions leads to a *normal* conditional rewrite relation. The reader can check that the whole system given in Sect. 2 is orthonormal.

We now prove that $\rightarrow_{\beta\cup\mathcal{B}}$ is shallow confluent (*i.e.* $\rightarrow^*_{\beta\cup\mathcal{B}_i}$ and $\rightarrow^*_{\beta\cup\mathcal{B}_j}$ commute for all $i, j \geq 0$) when $\mathcal{R}$ is orthonormal. The first point is that confluence of $\rightarrow_{\beta\cup\mathcal{B}_i}$ implies commutation of $\rightarrow^*_\beta$ and $\rightarrow^*_{\mathcal{B}_{i+1}}$. The proof is as in Sect. 3.1, except that in a rule $\boldsymbol{d} = \boldsymbol{c} \supset l \rightarrow r$, $\boldsymbol{c}$ are closed $\rightarrow_{\beta\cup\mathcal{B}}$-normal forms. The main Lemma concerns commutation of parallel relations of $\rhd_{\mathcal{B}_i}$ and $\rhd_{\mathcal{B}_j}$ for all $i, j \geq 0$. But here, we use a weak form of parallelization: $\rhd_{\mathcal{B}_i}$ is simply the parallel closure of $\rightarrow_{\mathcal{B}_i}$. The name of the Lemma is usual for this kind of result with rewriting (see [21]). Write $<_{mul}$ for the multiset extension of the usual ordering on naturals numbers.

Lemma 20 (Parallel Moves). *Let $\mathcal{R}$ be an orthonormal system. If $\{n,m\} <_{mul} \{i,j\}$ implies commutation of $\rightarrow^*_{\beta\cup\mathcal{B}_n}$ and $\rightarrow^*_{\beta\cup\mathcal{B}_m}$, then $\rhd_{\mathcal{B}_i}$ and $\rhd_{\mathcal{B}_j}$ commute.*

Proof Sketch. The key point is the commutation of $\rightarrow^*_{\beta\cup\mathcal{B}_n}$ and $\rightarrow^*_{\beta\cup\mathcal{B}_m}$ for $\{n,m\} <_{mul} \{i,j\}$. It implies that two rules whose respectives conditions are satisfied with $\rightarrow^*_{\beta\cup\mathcal{B}_i}$ and $\rightarrow^*_{\beta\cup\mathcal{B}_j}$ are not superposable at non-variable positions. The rest of the proof follows usual schemes (see Sect. 7.4 in [21]). □

Now, an induction on $<_{mul}$ provides the commutation of $\rightarrow_{\beta\cup\mathcal{B}_i}$ and $\rightarrow_{\beta\cup\mathcal{B}_j}$ for all $i,j \geq 0$. Shallow confluence immediately follows.

Theorem 21. *If $\mathcal{R}$ is an orthonormal system, then $\rightarrow_{\beta\cup\mathcal{B}}$ is shallow confluent.*

Hence, the relation $\rightarrow_{\beta\cup\mathcal{B}}$ induced by the system of Sect. 2 is confluent.

6 Conclusion

Our results are summarized in the following table.

§	Terms	Lhs	Rhs	Conditions	Result
3.1	$\mathcal{T}$	Linear	Applicative	Semi-Closed	$\rightarrow_{\mathcal{A}}$ Confluent $\Rightarrow$ $\rightarrow_{\mathcal{A}\cup\beta}$ Confluent
3.2	$\mathcal{U}$		Arity-Compliant & Algebraic	Arity-Compliant & Algebraic	idem
4.1	$\mathcal{R}$-stable	Linear	Algebraic	Semi-Closed & Algebraic	$\rightarrow_{\mathcal{A}}$ Confluent $\Rightarrow$ $\rightarrow_{\mathcal{B}\cup\beta}$ Confluent
4.2	$\mathcal{U}$		Arity-Compliant & Algebraic	Arity-Compliant & Algebraic	idem
5	$\mathcal{T}$	Linear		Orthonormal	$\rightarrow_{\mathcal{B}\cup\beta}$ Shallow Confluent

We provide detailed conditions to ensure modularity of confluence when combining β-reduction and conditional rewriting, either when the evaluation of conditions uses β-reduction or when it does not. This has useful applications on the high-level specification side and for enriching the conversion used in logical frameworks or proof assistants, while still preserving the confluence property.

These results lead us to the following remarks and further research points. The results obtained in Sect. 3 and 4 for the standard conditional rewrite systems extend to the case of oriented systems (hence to normal systems) and to the case of level-confluent natural systems. For natural systems, the proofs follow the same scheme, provided that level-confluence of $\rightarrow_{\mathcal{A}}$ is assumed. However, it would be interesting to know if this restriction can be dropped.

Problems arising from non left-linear rewriting are directly transposed to left-linear conditional rewriting. The semi-closure condition is sufficient to avoid this,

and it provides the counter part of left-linearity for unconditional rewriting. As a matter of a fact, it is well known that orthogonal standard conditional rewrite systems are not confluent, but confluence of orthogonal semi-closed standard systems holds. However, two remarks have to be made about this restriction. First, it would be interesting to know if it is a necessary condition and besides, to characterize a class of non semi-closed systems that can be translated into equivalent semi-closed ones. Second, semi-closed terminating standard systems behave like normal systems. But normal systems can be easily translated in equivalent non-conditional systems. Moreover such a translation preserves good properties such as left-linearity and non-ambiguity. As many of practical uses of rewriting rely on terminating systems, semi-closed standard systems may be in practice essentially an intuitive way to design rewrite systems that can be then efficiently implemented by non-conditional rewriting.

An interesting extension of this work consists in adapting to conditional rewriting the axiomatization and the results of [23]. This should leads to a generalization of the higher-order conditional systems of [1].

Acknowledgments. We are quite grateful to the anonymous referees for their constructive and accurate comments and suggestions.

References

[1] J. Avenhaus and C. Loría-Sáenz. Higher order conditional rewriting and narrowing. In *Proceedings of the 1st International Conference on Constraints in Computational Logics*, volume 845 of *LNCS*, pages 269–284. Springer Verlag, 1994.

[2] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of Strong Normalisation and Confluence in the Algebraic λ-Cube. *Journal of Functional Programming*, 7(6):613–660, November 1997.

[3] H.P. Barendregt. *The Lambda-Calculus, its syntax and semantics.* Studies in Logic and the Foundation of Mathematics. North Holland, 1984. Second edition.

[4] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Principles of Programming Languages, New Orleans, USA*. ACM, 2003.

[5] F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures In Computer Science*, 15(1):37–92, 2005.

[6] F. Blanqui, C. Kirchner, and C. Riba. On the confluence of lambda-calculus with conditional rewriting. HAL technical report, Oct 2005.

[7] V. Breazu-Tannen. Combining algebra and higher-order types. In *3rd IEEE Symposium on Logic in Computer Science Edinburg (UK)*, july 1988.

[8] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic confluence. *Information and Computation*, 114(1):1–29, October 1994.

[9] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[10] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. North-Holland, 1990.

[11] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.

[12] D.J. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. *Information and Computation*, 101(2):251–267, December 1992.

[13] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003.

[14] Bernhard Gramlich. On termination and confluence properties of disjoint and constructor-sharing conditional rewrite systems. *Theoretical Computer Science*, 165(1):97–131, September 1996.

[15] J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proceedings of LICS'91.*

[16] J.W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Center Tracts*. CWI, 1980. PhD Thesis.

[17] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *TCS*, 121:279–308, 1993.

[18] A. Middeldorp. Completeness of Combinations of Conditional Constructor Systems. *Journal of Symbolic Computation*, 17(1):3–21, January 1994.

[19] F. Müller. Confluence of the lambda calculus with left linear algebraic rewriting. *Information Processing Letters*, 41:293–299, 1992.

[20] T. Nipkow. Higher-order critical pairs. In *Proceedings of LICS'91.*

[21] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, April 2002.

[22] M. Takahashi. Lambda-calculi with conditional rules. In *TLCA'93*, LNCS, pages 406–417. Springer-Verlag, 1993.

[23] V. van Oostrom and F. van Raamsdonk. Weak orthogonality implies confluence: the higher-order case. In *LFCS'94*, volume 813 of *LNCS*, 1994.

Guessing Attacks and the Computational Soundness of Static Equivalence

Martín Abadi[1], Mathieu Baudet[2], and Bogdan Warinschi[3]

[1] University of California, Santa Cruz
[2] LSV, CNRS & INRIA Futurs projet SECSI & ENS Cachan, France
[3] Loria, INRIA, Nancy, France

Abstract. The indistinguishability of two pieces of data (or two lists of pieces of data) can be represented formally in terms of a relation called static equivalence. Static equivalence depends on an underlying equational theory. The choice of an inappropriate equational theory can lead to overly pessimistic or overly optimistic notions of indistinguishability, and in turn to security criteria that require protection against impossible attacks or—worse yet—that ignore feasible ones. In this paper, we define and justify an equational theory for standard, fundamental cryptographic operations. This equational theory yields a notion of static equivalence that implies computational indistinguishability. Static equivalence remains liberal enough for use in applications. In particular, we develop and analyze a principled formal account of guessing attacks in terms of static equivalence.

1 Introduction

In the study of security, it is frequent to reason about whether two pieces of data can be distinguished by an observer. For example, the pieces of data might be two encrypted messages, and the observer an attacker that attempts to learn something about the underlying cleartexts by analyzing the encrypted messages. The two encrypted messages are indistinguishable if, no matter how the attacker operates on them, it cannot discern any meaningful difference. The encrypted messages may however be different—for instance, they may look like different random numbers.

Formally, indistinguishability can be represented in terms of a relation called static equivalence [4]. Roughly, two terms (and, more generally, two lists of terms) are statically equivalent when they satisfy all the same equations. This relation is essentially a special case of the observational equivalence relation of process calculi. It is simpler than observational equivalence in that it does not allow for continued interaction between a system and an observer: the observer gets data once and then conducts experiments on its own. Nevertheless, observational equivalence can be reduced to a combination of static equivalence and usual bisimulation requirements [4, 5, 15].

Static equivalence depends on an underlying equational theory. The choice of an inappropriate equational theory can lead to overly pessimistic or optimistic

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 398–412, 2006.
© Springer-Verlag Berlin Heidelberg 2006

notions of indistinguishability, and in turn to security criteria that require protection against impossible attacks or—worse yet—that ignore feasible ones.

In this paper, we define an equational theory for standard, fundamental cryptographic operations, and we justify and apply the resulting concept of static equivalence. These operations include various flavors of encryption and decryption. Static equivalence in this theory implies computational indistinguishability. In other words, if the formal notion of static equivalence indicates that two pieces of data are indistinguishable, then no computationally feasible experiment can tell those two pieces of data apart. (This property is a soundness theorem. Although it is less important, we have also explored completeness, but we omit its discussion here; see however [28, 9].) Our notion of computational feasibility is based on the sorts of assumptions typically employed in complexity-theoretic cryptography. It includes certain assumptions on the security properties of the cryptographic operations; those assumptions appear reasonable and fairly standard, but so do others, and picking satisfactory ones is somewhat delicate.

While static equivalence is conservative enough to exclude feasible attacks, it also remains liberal enough for use in applications. In particular, we develop a formal account of guessing attacks (e.g., [23, 24, 13, 32, 31]) in terms of static equivalence. Since guessing attacks constitute a significant threat against protocols that rely on passwords and other weak secrets, the recent literature contains several studies of guessing attacks, with both formal and computational approaches (e.g., [27, 19, 18, 17, 10, 12, 16, 22, 25, 21]). Formal approaches are attractive because of their relative simplicity, which often enables automation. On the other hand, formal approaches are rather varied and sometimes ad hoc. Fortunately, it has been suggested that a formulation of guessing attacks could be based on static equivalence [17, 20]. We believe that this idea has a number of virtues. It leads to a crisp definition, it is fairly independent of specific choices of cryptographic operations, and it extends nicely to general process calculi. To date, however, this idea has not been worked out fully, in the setting of an appropriate equational theory. We aim to address this gap.

A related, frequent shortcoming of formal analyses is the lack of computational justifications. This lack allows the possibility that a protocol is safe against attacks formally, but that a feasible attack exists nonetheless. An active line of recent work aims to address such shortcomings, by defining and proving computational soundness results for formal methods (e.g., [6, 8, 29, 26]). That line of research includes a computational study of static equivalence [11]; the theories considered there do not include the one that we define in this paper (in part because those theories do not model probabilistic encryption functions, nor encryption under weak keys) and have not provided a satisfactory account of guessing attacks, but they are an important piece of the context of this work. That line of research also includes a study of guessing attacks, with an ad hoc formal definition of those attacks [7]. In this paper we build on that previous work, and go beyond it.

The next section, Section 2, presents a formal model: it defines sorted terms, an equational theory for them, and the corresponding notion of static

equivalence. Section 3 interprets the syntax of the formal model in a computational universe; it includes cryptographic assumptions. Section 4 establishes the computational soundness of static equivalence for the equational theory. Section 5 applies our results to the study of guessing attacks. Section 6 concludes. Because of space constraints, we omit cryptographic constructions, a decision procedure for static equivalence, proofs, and additional details; these are included in an extended version of this paper [1].

2 Abstract Model

In order to represent cryptographic messages in an abstract way, we use terms over a many-sorted signature, equipped with an equational theory.

2.1 Sorts and Terms

The set of *sorts (or types)* that we consider is defined by the following grammar:

τ ::=

$\mid$	$SKey$	symmetric keys
$\mid$	$EKey$	(public) encryption keys
$\mid$	$DKey$	(private) decryption keys
$\mid$	$Data$	passwords and other data
$\mid$	$Coins$	coins for encryption
$\mid$	$Pair[\tau_1, \tau_2]$	pairs of messages
$\mid$	$SCipher[\tau]$	symmetric encryptions of messages of type τ
$\mid$	$ACipher[\tau]$	asymmetric encryptions of messages of type τ

The set of *(well-sorted) terms*, written $S, T, U, V, \ldots$, is built from an infinite number of variables x, y, ... and names a, b, n, r, k, sk, pk, ... for each sort, with the following function symbols:

$\mathsf{enc}_\tau : \tau \times Data \to \tau$	encryption under data
$\mathsf{dec}_\tau : \tau \times Data \to \tau$	decryption with data
$\mathsf{penc}_\tau : \tau \times EKey \times Coins \to ACipher[\tau]$	public-key encryption
$\mathsf{pdec}_\tau : ACipher[\tau] \times DKey \to \tau$	private-key decryption
$\mathsf{pub} : DKey \to EKey$	public-key extraction
$\mathsf{pdec_success}_\tau : ACipher[\tau] \times DKey \to Data$	domain predicate for private-key decryption
$\mathsf{senc}_\tau : \tau \times SKey \times Coins \to SCipher[\tau]$	symmetric encryption
$\mathsf{sdec}_\tau : SCipher[\tau] \times SKey \to \tau$	symmetric decryption
$\mathsf{sdec_success}_\tau : SCipher[\tau] \times SKey \to Data$	domain predicate for symmetric decryption
$\mathsf{pair}_{\tau_1,\tau_2} : \tau_1 \times \tau_2 \to Pair[\tau_1, \tau_2]$	pairing
$\mathsf{fst}_{\tau_1,\tau_2} : Pair[\tau_1, \tau_2] \to \tau_1$	first projection
$\mathsf{snd}_{\tau_1,\tau_2} : Pair[\tau_1, \tau_2] \to \tau_2$	second projection
$0, 1 : Data$	boolean constants
$\mathsf{w}, \mathsf{c}_0, \mathsf{c}_1 \ldots : Data$	additional data constants

Encryption and decryption symbols may not be available for all sorts τ. We let T_{penc} be the set of types τ for which the symbols penc_τ, pdec_τ, and $\mathsf{pdec_success}_\tau$ are available, and define T_{senc} and T_{enc} analogously. We assume that pairs are not encrypted under data values, that is, $T_{\mathsf{enc}} \cap \{Pair[\tau_1, \tau_2]\}_{\tau_1,\tau_2} = \emptyset$; pairs may however be encrypted with enc component by component.

Our function symbols represent encryption and decryption functions and auxiliary operations. The first two functions (enc_τ and dec_τ) are to be used with data values as keys; the data values may be the constant symbols of the grammar, which may represent the passwords in a dictionary. (In contrast, fresh names may represent strong keys; the scoping rules justify the respective uses of constant symbols and names.) The fact that enc_τ does not take a parameter of type *Coins* relates to the difficulties with probabilistic password-based encryption [7]. Moreover, the language provides no direct way for the attacker to check that a value results from applying enc_τ with a particular key. Such properties are essential for thwarting guessing attacks in practice (for example, in the EKE protocol [13]). The remaining functions are fairly standard; they include functions for public-key and symmetric encryption (penc_τ and senc_τ), which are probabilistic in the sense that they take a parameter of type *Coins*.

We often omit type annotations on function symbols. For instance, provided that S, T, and U have type *Data*, we may write $\mathsf{pair}(\mathsf{enc}(S,T),U)$ instead of $\mathsf{pair}_{Data,Data}(\mathsf{enc}_{Data}(S,T),U)$. In addition, we sometimes use the abbreviations $\{S\}_T$ for $\mathsf{enc}(S,T)$, $\{S\}^r_{\mathsf{pub}(sk)}$ for $\mathsf{penc}(S,\mathsf{pub}(sk),r)$, and $\{S\}^r_k$ for $\mathsf{senc}(S,k,r)$.

We write $\mathrm{var}(T)$ and $\mathrm{names}(T)$ for the sets of variables and names that occur in a term T. We extend the notation to tuples and sets of terms. A term T is *ground* or *closed* when $\mathrm{var}(T) = \emptyset$. We write $\sigma = \{x_1 \mapsto T_1, \ldots, x_n \mapsto T_n\}$ for a substitution, and let $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$, $\mathrm{var}(\sigma) = \mathrm{var}(T_1, \ldots, T_n)$, and $\mathrm{names}(\sigma) = \mathrm{names}(T_1, \ldots, T_n)$. A substitution σ is *ground* or *closed* when $\mathrm{var}(\sigma) = \emptyset$. We consider only well-sorted substitutions (that is, for each i, $T_i = x_i\sigma$ has the same sort as x_i).

2.2 Equational Theory

We model the semantics of the cryptographic primitives by equipping terms with an equational theory, that is, a reflexive, symmetric, transitive relation, stable by (well-sorted) substitutions of terms for variables and (in this case) for names, and stable by application of contexts. Specifically, we consider the equational theory $=_E$ generated by the following equations:

$$
\begin{array}{rr}
\mathsf{dec}_\tau(\mathsf{enc}_\tau(x,y),y) = x & \mathsf{enc}_\tau(\mathsf{dec}_\tau(x,y),y) = x \\
\mathsf{pdec}_\tau(\mathsf{penc}_\tau(x,\mathsf{pub}(y),z),y) = x & \mathsf{pdec_success}_\tau(\mathsf{penc}_\tau(x,\mathsf{pub}(y),z),y) = 1 \\
\mathsf{sdec}_\tau(\mathsf{senc}_\tau(x,y,z),y) = x & \mathsf{sdec_success}_\tau(\mathsf{senc}_\tau(x,y,z),y) = 1 \\
\mathsf{fst}_{\tau_1,\tau_2}(\mathsf{pair}_{\tau_1,\tau_2}(x,y)) = x & \mathsf{snd}_{\tau_1,\tau_2}(\mathsf{pair}_{\tau_1,\tau_2}(x,y)) = y \\
\mathsf{pair}_{\tau_1,\tau_2}(\mathsf{fst}_{\tau_1,\tau_2}(x),\mathsf{snd}_{\tau_1,\tau_2}(x)) = x &
\end{array}
$$

where the symbols x, y, and z represent variables of the appropriate sorts. Most of the equations are fairly standard. The only surprise may be the inclusion of $\mathsf{enc}_\tau(\mathsf{dec}_\tau(x,y),y) = x$, without which an attacker that sees x and guesses y

might confirm whether x is a ciphertext encrypted under y by decrypting x with y, reencrypting with y, and comparing the result to x; the equation implies that the comparison always succeeds, whether the guess was correct or not. So, for instance, $\mathsf{enc}_\tau(n, \mathsf{c}_0)$ and $\mathsf{enc}_\tau(n, \mathsf{c}_1)$ are indistinguishable when n is a fresh name of sort τ. Such consequences of the equation are important for the security of protocols that rely on weak secrets. Moreover, the equation holds in many reasonable implementations, in particular those based on keyed permutations.

When oriented from left to right, the equations above form a convergent rewriting system that we call $\mathcal{R}$.

2.3 Frames and Static Equivalence

Frames represent sets of messages available to an observer (for example, because they were sent over a public network) [4]. More precisely, a *frame* is an expression $\varphi = \nu\tilde{n}.\{x_1 = T_1, \ldots, x_n = T_n\}$ where $\tilde{n}$ is a set of *restricted* names, and each T_i is a closed term of the same sort as x_i. For simplicity, we require (without loss of generality) that every name in use be restricted, that is, $\tilde{n} = \mathrm{names}(T_1, \ldots, T_n)$. A name k may still be disclosed explicitly, for instance by a dedicated mapping $x_i = k$. Therefore, we tend to omit the binders $\nu\tilde{n}$, and identify a frame φ with its *underlying substitution* $\{x_1 \mapsto T_1, \ldots, x_n \mapsto T_n\}$.

A closed term T is *deducible* from a frame φ if there exists a term M with $\mathrm{var}(M) \subseteq \mathrm{dom}(\varphi)$ and $\mathrm{names}(M) \cap \mathrm{names}(\varphi) = \emptyset$ such that $M\varphi =_E T$ [2, 3].

Two frames φ_1 and φ_2 such that $\mathrm{dom}(\varphi_1) = \mathrm{dom}(\varphi_2)$ are *statically equivalent* (written $\varphi_1 \approx_E \varphi_2$) if, for every pair of terms (M, N) such that $\mathrm{var}(M, N) \subseteq \mathrm{dom}(\varphi_1)$ and $\mathrm{names}(M, N) \cap \mathrm{names}(\varphi_1, \varphi_2) = \emptyset$, it holds that $M\varphi_1 =_E N\varphi_2$ if and only if $M\varphi_2 =_E N\varphi_2$. Proving static equivalence may not be easy. Fortunately, efficient methods exist in many cases (e.g., [2, 14]). In particular, static equivalence is decidable in polynomial time for unsorted convergent subterm theories [2]; we expect that this result carries over to sorted convergent subterm theories such as $=_E$. We have an alternative decision procedure for the static equivalences that are the subject of our main theorem (see Section 4).

We close this section with a few examples of equivalences and inequivalences under the theory E:

$$\{x = \{0\}_k^r\} \approx_E \{x = \{1\}_k^r\} \tag{1}$$

$$\{x = \{0\}_k^r,\ \ y = \{0\}_k^{r'}\} \approx_E \{x = \{1\}_k^r,\ \ y = \{0\}_{k'}^{r'}\} \tag{2}$$

$$\{x = \{n\}_{\mathsf{w}},\ \ y = \{m\}_{\mathsf{w}}\} \approx_E \{x = a_1,\ \ y = a_2\} \tag{3}$$

$$\{x = \{\{n\}_{\mathsf{w}}\}_{\mathsf{w}},\ \ y = \{m\}_{\mathsf{w}}\} \approx_E \{x = a_1,\ \ y = a_2\} \tag{4}$$

$$\{x = \{\{0\}_{\mathsf{pub}(sk)}^{r_1}\}_{\mathsf{w}},\ \ y = \{0\}_{\mathsf{pub}(sk)}^{r_2}\} \approx_E \{x = a_1,\ \ y = a_2\} \tag{5}$$

$$\{x = \{\{0\}_{\mathsf{pub}(sk)}^{r_1}\}_{\mathsf{w}},\ \ y = \{0\}_{\mathsf{pub}(sk)}^{r_1}\} \approx_E \{x = \{a_1\}_{\mathsf{w}},\ \ y = a_1\} \tag{6}$$

$$\{x = \{\{n\}_k^{r_1}\}_{\mathsf{w}},\ \ y = k\} \not\approx_E \{x = a_1,\ \ y = k\} \tag{7}$$

Examples (1) and (2) are simple examples about symmetric encryptions under strong keys, illustrating that those encryptions hide plaintexts and also equalities of plaintexts or keys across encryptions. Examples (3) and (4) illustrate that

encryptions of fresh names under a constant w (intuitively, under a weak secret) can look like fresh names. The values of x and y are two such encryptions—and the former is in fact a double encryption in example (4)—with unrelated underlying names. Example (5) resembles example (4); it illustrates that an encryption of a public-key ciphertext $\{0\}^{r_1}_{\mathsf{pub}(sk)}$ under w can look like a fresh name. In examples (3)–(5), the plaintexts being encrypted are not otherwise available to the observer, though somewhat related plaintexts may be (as the values of the variable y). Example (6) treats a case in which the observer also obtains the plaintext being encrypted, through y; in that case, the observer can see a relation between the value of x and the value of y, namely that the former is an encryption of the latter under w. Example (7) indicates that the observer that is given k can distinguish $\{\{n\}^{r_1}_k\}_{\mathsf{w}}$ from a fresh name; intuitively, after decrypting with w, the adversary can tell if what it sees is a ciphertext under k or not, since the success of shared-key decryption is detectable.

3 Implementation

In this section we interpret the syntax of the formal model in a computational universe. We also discuss cryptographic assumptions on which the implementation relies.

3.1 Interpreting the Syntax

Next we detail the mapping from terms to distribution ensembles over bit-strings.

Encryption schemes. The mapping uses a public-key encryption scheme $\Pi^p = (\mathcal{K}^p, \mathcal{E}^p, \mathcal{D}^p)$ and a symmetric encryption scheme $\Pi^s = (\mathcal{K}^s, \mathcal{E}^s, \mathcal{D}^s)$. It also uses a symmetric, deterministic, type-preserving encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. (The definition of type preserving is given below.) In each of these triples, the first component is a key-generation function, the second an encryption function, and the third a decryption function. We write η for a security parameter. For each η, we write $k \stackrel{R}{\longleftarrow} \mathcal{K}_\eta$ and $k \stackrel{R}{\longleftarrow} \mathcal{K}^s_\eta$ for the process of generating an encryption key k for Π and Π^s, respectively, and similarly we write $(pk, sk) \stackrel{R}{\longleftarrow} \mathcal{K}^p_\eta$ for the process of generating a pair (pk, sk) of encryption and decryption keys for Π^p. As usual, the encryption functions $\mathcal{E}^p$ and $\mathcal{E}^s$ are randomized; we write $\mathcal{E}^p(m, k, r)$ and $\mathcal{E}^s(m, k, r)$ for public-key and symmetric encryptions, respectively, of message m under encryption key k with random coins r. We write $c \stackrel{R}{\longleftarrow} \mathcal{E}^p(m, k)$ and $c \stackrel{R}{\longleftarrow} \mathcal{E}^s(m, k)$ for the corresponding encryption processes, using fresh random coins. We assume that the set of keys for Π is of the form $\{0,1\}^{\alpha_1(\eta)}$, and that the set of coins for Π^s and Π^p is $\{0,1\}^{\alpha_2(\eta)}$, where the functions $\alpha_1(\eta)$ and $\alpha_2(\eta)$ are polynomially bounded and at least linearly increasing.

We say that Π is *type-preserving* when, for every $\tau \in T_{\mathsf{enc}}$, encryption and decryption by Π map $[\![\tau]\!]_\eta$—the set of bit-strings that corresponding to the type τ—to itself.

Sorts, functions, and random drawings. For each value of the security parameter η, the concrete meaning of sorts and terms is characterized (much as in [11]) by:

- for each sort τ, a carrier set $[\![\tau]\!]_\eta$;
- for each function symbol $f : \tau_1 \times \ldots \times \tau_n \to \tau$, a function $[\![f]\!]_\eta : [\![\tau_1]\!]_\eta \times \ldots \times [\![\tau_n]\!]_\eta \to [\![\tau]\!]_\eta$;
- for each sort τ, a procedure written $e \xleftarrow{R} [\![\tau]\!]_\eta$ for drawing a random element e from $[\![\tau]\!]_\eta$, according to a distribution written $(\xleftarrow{R} [\![\tau]\!]_\eta)$.

We require that no element in $[\![\tau]\!]_\eta$ has probability 0 according to $(\xleftarrow{R} [\![\tau]\!]_\eta)$, that the probability of collision for $(\xleftarrow{R} [\![\tau]\!]_\eta)$ is negligible (that is, asymptotically smaller than any inverse polynomial), and that all the operations mentioned are computable in probabilistic polynomial time (PPTIME) in the complexity parameter. These conditions are ensured by the construction below and the properties of secure encryption schemes (defined in the next subsection).

The carrier set $[\![\tau]\!]_\eta$ of a type τ is defined inductively:

$$\begin{aligned}
[\![SKey]\!]_\eta &= \text{“}SKey\text{”} \parallel \{\text{symmetric keys for } \Pi^s(\eta)\} \\
[\![EKey]\!]_\eta &= \text{“}EKey\text{”} \parallel \{\text{public keys for } \Pi^p(\eta)\} \\
[\![DKey]\!]_\eta &= \text{“}DKey\text{”} \parallel \{\text{private keys for } \Pi^p(\eta)\} \\
[\![Data]\!]_\eta &= \text{“}Data\text{”} \parallel \{0,1\}^{\alpha_1(\eta)} \\
[\![Coins]\!]_\eta &= \text{“}Coins\text{”} \parallel \{0,1\}^{\alpha_2(\eta)} \\
[\![Pair[\tau_1,\tau_2]]\!]_\eta &= \text{“}Pair\text{”} \parallel [\![\tau_1]\!]_\eta \parallel [\![\tau_2]\!]_\eta \\
[\![SCipher[\tau]]\!]_\eta &= \text{“}SCipher\text{”} \parallel \tau \parallel \{\text{ciphertexts of } \Pi^s(\eta)\} \\
[\![ACipher[\tau]]\!]_\eta &= \text{“}ACipher\text{”} \parallel \tau \parallel \{\text{ciphertexts of } \Pi^p(\eta)\}
\end{aligned}$$

where $\parallel$ denotes the concatenation of bit-strings (applied by extension on sets of bit-strings), and we assume an encoding of identifiers for types τ into bit-strings.

The meaning of function symbols is as follows:

- Symbols $\mathsf{pair}_{\tau_1,\tau_2}$, $\mathsf{fst}_{\tau_1,\tau_2}$, and $\mathsf{snd}_{\tau_1,\tau_2}$ are implemented on bit-strings by tagged concatenation and projections, as one might expect.
- Constants w, c_0, c_1, ... are mapped to arbitrary PPTIME-computable sequences of bit-strings of length $\alpha_1(\eta)$, prefixed with the tag “*Data*”; 0 and 1 are mapped respectively to “*Data*” $\parallel 0^{\alpha_1(\eta)}$ and “*Data*” $\parallel 1^{\alpha_1(\eta)}$.
- For every $\tau \in T_{\mathsf{penc}}$, the implementations of penc_τ, pdec_τ, and $\mathsf{pdec_success}_\tau$ are defined by:

$$[\![\mathsf{penc}_\tau]\!]_\eta(m, \text{“}EKey\text{”}\|pk, \text{“}Coins\text{”}\|r) = \text{“}ACipher\text{”}\|\tau\|\mathcal{E}^p_\tau(m, pk, r)$$

$$[\![\mathsf{pdec}_\tau]\!]_\eta(m, \text{“}DKey\text{”}\|sk) = \begin{cases} \mathcal{D}^p(c, sk) & \text{if } m = \text{“}ACipher\text{”}\|\tau\|c \text{ and the} \\ & \text{decryption } \mathcal{D}^p(c, sk) \text{ succeeds} \\ \langle\text{any value}\rangle & \text{otherwise} \end{cases}$$

$$[\![\mathsf{pdec_success}_\tau]\!]_\eta(m, \text{“}DKey\text{”}\|sk) = \text{“}Data\text{”} \parallel \begin{cases} 1^{\alpha_1(\eta)} & \text{if } m = \text{“}ACipher\text{”}\|\tau\|c \text{ and the decryption } \mathcal{D}^p(c, sk) \text{ succeeds} \\ 0^{\alpha_1(\eta)} & \text{otherwise} \end{cases}$$

The implementations of senc_τ, sdec_τ, and $\mathsf{sdec_success}_\tau$, for $\tau \in T_{\mathsf{senc}}$, are defined similarly.
– For every $\tau \in T_{\mathsf{enc}}$, the implementations of enc_τ and dec_τ are defined by:

$$\begin{aligned}[\![\mathsf{enc}_\tau]\!]_\eta(m, \text{“}Data\text{”}\|k) &= \mathcal{E}(m,k)\\ [\![\mathsf{dec}_\tau]\!]_\eta(c, \text{“}Data\text{”}\|k) &= \mathcal{D}(c,k)\end{aligned}$$

We assume that $\mathcal{E}(\cdot,k)$ and $\mathcal{D}(\cdot,k)$ are inverse bijections from $[\![\tau]\!]_\eta$ to itself. In particular, tags are left unchanged by these functions.

The drawing of random values of type τ ($e \xleftarrow{R} [\![\tau]\!]_\eta$) is defined by induction on τ (with, in addition, the appropriate tags in each case):

– When τ is one of $SKey$, $EKey$, and $DKey$, use the dedicated key generation algorithm, respectively $\mathcal{K}^s$, $\mathsf{fst}(\mathcal{K}^p)$, and $\mathsf{snd}(\mathcal{K}^p)$.
– When τ is $Data$ or $Coins$, use the uniform distribution over $[\![\tau]\!]_\eta$.
– When $\tau = Pair[\tau_1,\tau_2]$, recursively draw random elements in $[\![\tau_1]\!]_\eta$ and $[\![\tau_2]\!]_\eta$, then tag and concatenate them.
– When τ is $SCipher[\tau]$ or $ACipher[\tau]$, encrypt a random element in $[\![\tau]\!]_\eta$ with a fresh random key of the appropriate kind.

Interpreting terms and frames. Given η, we associate with each frame $\varphi = \nu\widetilde{n}.\{x_1 = T_1,\ldots,x_n = T_n\}$ a distribution $[\![\varphi]\!]_\eta$ defined by the following procedure for computing a sample $\phi \xleftarrow{R} [\![\varphi]\!]_\eta$:

1. for each name of sort τ that occurs in φ, draw a value $\widehat{a} \xleftarrow{R} [\![\tau]\!]_\eta$;
2. compute the value $\widehat{T_i}$ of each closed term T_i, recursively:

$$\text{for every function symbol } f, \quad \widehat{f(S_1,\ldots,S_n)} = [\![f]\!]_\eta(\widehat{S_1},\ldots,\widehat{S_1})$$

3. let the resulting *concrete frame* be $\phi = \{x_1 = \widehat{T_1},\ldots,x_n = \widehat{T_n}\}$.

We define the notation $[\![_]\!]_\eta$ for closed terms and tuples of closed terms similarly. We may write $[\![\varphi]\!]_{\eta,\, a_1\mapsto e_1,\,\ldots,\, a_n\mapsto e_n}$ so as to specify the values for names, and $[\![\varphi]\!]_{\eta,\, \mathsf{c}_1\mapsto e_1,\,\ldots,\, \mathsf{c}_n\mapsto e_n}$ so as to specify the values of the constants $\mathsf{c}_1, \ldots, \mathsf{c}_n$. We write $[\![\varphi]\!]$ for the *ensemble* (family of distributions) $([\![\varphi]\!]_\eta)_\eta$. We identify a single-valued (Dirac) distribution with its unique value.

Indistinguishability. Two ensembles $D^1 = (D^1_\eta)_\eta$ and $D^2 = (D^2_\eta)_\eta$ are *indistinguishable*, written $D^1 \approx D^2$, when, for every PPTIME adversary A, the function

$$\mathrm{Adv}_A(\eta) = \mathbb{P}\left[e \xleftarrow{R} D^1_\eta : A(e) = 1\right] - \mathbb{P}\left[e \xleftarrow{R} D^2_\eta : A(e) = 1\right]$$

is negligible.

3.2 Cryptographic Assumptions

We use symmetric and asymmetric encryption schemes that satisfy a notion of security related to type-0 and type-1 security [6]. Essentially, we require that for each type τ, the encryption function restricted to elements of $[\![\tau]\!]$ reveal no

information about the key used for encryption and hide all partial information about underlying plaintexts—except for their belonging to the carrier set $[\![\tau]\!]$.

Definition 1. *Let $\Pi^s = (\mathcal{K}^s, \mathcal{E}^s, \mathcal{D}^s)$ be a symmetric encryption scheme. For each security parameter η and type $\tau \in T_{\mathsf{senc}}$, we consider the following experiment, with a two-stage PPTIME adversary $A = (A_1, A_2)$:*

- *a key k is generated via $k \stackrel{R}{\longleftarrow} \mathcal{K}^s(\eta)$;*
- *A_1 is provided access to an oracle $\mathcal{E}^s(\cdot, k)$, that is, A_1 may submit messages m to the oracle and receives in return corresponding encryptions $\mathcal{E}^s(m, k)$;*
- *then A_1 outputs a challenge message $m^* \in [\![\tau]\!]_\eta$ together with some state information st;*
- *a bit $b \stackrel{R}{\longleftarrow} \{0,1\}$ is selected at random; if $b = 0$, we let c be a (tagged) encryption of m^* under k, that is, $c \stackrel{R}{\longleftarrow}$ "SCipher"$\|\tau\|\mathcal{E}^s(m^*, k)$; otherwise, we let c be a (tagged) encryption of a random element of τ under a random key, that is, $c \stackrel{R}{\longleftarrow} [\![SCipher[\tau]]\!]_\eta$;*
- *A_2 is given c and st, and outputs a bit b'.*

*The adversary A is successful if $b' = b$. The advantage of A is defined by $\mathrm{Adv}^{\tau}_{\Pi^s,A}(\eta) = \Pr[A \text{ is successful}] - \frac{1}{2}$. We say that Π^s is T_{senc}-*secure *if for all PPTIME adversaries A and all $\tau \in T_{\mathsf{senc}}$, the function $\mathrm{Adv}^{\tau}_{\Pi^s,A}(\cdot)$ is negligible.*

Definition 2. *Let $\Pi^p = (\mathcal{K}^p, \mathcal{E}^p, \mathcal{D}^p)$ be an asymmetric encryption scheme. For each security parameter η and type $\tau \in T_{\mathsf{penc}}$, we consider the following experiment, with a two-stage PPTIME adversary $A = (A_1, A_2)$:*

- *a pair of encryption/decryption keys (pk, sk) is generated via $(pk, sk) \stackrel{R}{\longleftarrow} \mathcal{K}^p(\eta)$, and A_1 is given pk;*
- *A_1 outputs a challenge message $m^* \in [\![\tau]\!]_\eta$ together with some state information st;*
- *a bit $b \stackrel{R}{\longleftarrow} \{0,1\}$ is selected at random; if $b = 0$, we let c be a (tagged) encryption of m^* under pk, that is, $c \stackrel{R}{\longleftarrow}$ "ACipher"$\|\tau\|\mathcal{E}^p(m^*, pk)$; otherwise, we let c be a (tagged) encryption of a random element of τ under a random public key, that is, $c \stackrel{R}{\longleftarrow} [\![ACipher[\tau]]\!]_\eta$;*
- *A_2 is given c and st, and outputs a bit b'.*

*The adversary $A = (A_1, A_2)$ is successful if $b' = b$. The advantage of A is defined by $\mathrm{Adv}^{\tau}_{\Pi^p,A}(\eta) = \Pr[A \text{ is successful}] - \frac{1}{2}$. We say that Π^p is T_{penc}-*secure *if for all PPTIME adversaries A and all $\tau \in T_{\mathsf{penc}}$, the function $\mathrm{Adv}^{\tau}_{\Pi^p,A}(\cdot)$ is negligible.*

Our notion of security for encryption schemes that use data values (such as passwords) as keys is less standard—and there is not yet a standard notion in the area:

Definition 3. *Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric, deterministic, type-preserving encryption scheme such that the set of keys is $\{0,1\}^{\alpha_1(\eta)}$ for each η.*

1. **Real-or-Random security (T_{enc}-RoR):** *For each security parameter η and type $\tau \in T_{\mathsf{enc}}$, we consider the following experiment, with a two-stage PPTIME adversary $A = (A_1, A_2)$:*
   - *a key k is generated via $k \xleftarrow{R} \mathcal{K}(\eta)$;*
   - *A_1 is provided access to an oracle $\mathcal{E}(\cdot, k)$, that is, A_1 may submit (tagged) messages m to the oracle and receives in return corresponding (tagged) encryptions $\mathcal{E}(m, k)$;*
   - *then A_1 submits a challenge message $m^* \in [\![\tau]\!]_\eta$ and some state information st;*
   - *a bit $b \xleftarrow{R} \{0,1\}$ is selected at random; if $b = 0$, we let c be the (tagged) encryption of m^* under k, that is, $c = \mathcal{E}(m^*, k)$; otherwise, we let $c \xleftarrow{R} [\![\tau]\!]_\eta$ be a random element from $[\![\tau]\!]_\eta$;*
   - *A_2 is given c and st, and outputs a bit b'.*

   The adversary A is successful if $b' = b$ and *the challenge message m^* is different from all the messages m submitted by A to the encryption oracle. The advantage of A is $\mathrm{Adv}^{\tau}_{RoR,\Pi,A}(\eta) = \Pr[A \text{ is successful}] - \frac{1}{2}$. We say that Π is* T_{enc}-RoR secure *if for all PPTIME adversaries A and all $\tau \in T_{\mathsf{enc}}$, the function $\mathrm{Adv}^{\tau}_{RoR,\Pi,A}(\cdot)$ is negligible.*

2. **Encryption under passwords or other data values (T_{enc}-Pwd):** *For each security parameter η and type $\tau \in T_{\mathsf{enc}}$, we consider the following experiment, with a two-stage PPTIME adversary $A = (A_1, A_2)$:*
   - *A_1 outputs a key $k \in \{0,1\}^{\alpha_1(\eta)}$ and some state information st;*
   - *a bit $b \xleftarrow{R} \{0,1\}$ is selected at random; if $b = 0$, we let c be the (tagged) encryption of some random element under k, that is, $m \xleftarrow{R} [\![\tau]\!]_\eta$ and $c = \mathcal{E}(m, k)$; otherwise, we let $c \xleftarrow{R} [\![\tau]\!]_\eta$ be a random element from $[\![\tau]\!]_\eta$;*
   - *A_2 is given c and st, and outputs a bit b'.*

   The adversary A is successful if $b' = b$. The advantage of A is defined by $\mathrm{Adv}^{\tau}_{Pwd,\Pi,A}(\eta) = \Pr[A \text{ is successful}] - \frac{1}{2}$. We say that Π is a T_{enc}-Pwd secure *if for all PPTIME adversaries A and all $\tau \in T_{\mathsf{enc}}$, the function $\mathrm{Adv}^{\tau}_{Pwd,\Pi,A}(\cdot)$ is negligible.*

Finally, Π is T_{enc}-secure *if it is both T_{enc}-RoR and T_{enc}-Pwd secure.*

Condition 1 (T_{enc}-RoR security) is a variant of IND-P1-C0 security [30, 11]. We require it because we allow enc to be used as a first-class encryption algorithm, that is, with strong keys (not just passwords). Without this condition, our main result remains true on frames which use only constants as keys for enc (much as in [7]).

Condition 2 (T_{enc}-Pwd security) addresses the security of passwords (or other data) when used as keys. Intuitively, it states that the encryption of a random value must be distributed like the value. A related previous condition [7] allows a possibly different distribution for the encryptions of random values and the values themselves. This difference is mostly due to the fact that we authorize multiple layers of encryptions with passwords (see example (4)).

Finally, an implementation with (Π^s, Π^p, Π) is $(T_{\mathsf{senc}}, T_{\mathsf{penc}}, T_{\mathsf{enc}})$-*secure* (or simply *secure*) if the three schemes Π^s, Π^p, and Π are, respectively, T_{senc}-secure, T_{penc}-secure, and T_{enc}-secure.

A possible secure implementation, using standard cryptographic tools, is outlined in the extended version of this paper [1].

4 Soundness of Static Equivalence

In this section we present our main soundness result. As usual (following [6]), this result requires a hypothesis that excludes encryption cycles, and also some other well-formedness conditions.

A *key position* in an expression is a position that corresponds to the argument U of a subterm of the form $\mathsf{pub}(U)$, or to the second argument V of a subterm $\mathsf{enc}_\tau(U, V)$, $\mathsf{penc}_\tau(U, V, W)$, or $\mathsf{senc}_\tau(U, V, W)$. An *encryption cycle* of a frame φ is a sequence of names k_0, k_1, ..., k_n of sort *Data*, *DKey*, and *SKey* such that $k_n = k_0$ and

> for each $0 \leq i \leq n-1$, there exists a subterm of φ of the form $\mathsf{enc}_\tau(U, V)$, $\mathsf{penc}_\tau(U, V, W)$, or $\mathsf{senc}_\tau(U, V, W)$ such that k_i is a subterm of U *not* in key position and k_{i+1} is a subterm of V.

For instance, the frame $\varphi_1 = \{x = \{sk_1\}_{k_2}^{r_1},\ y = \{k_2\}_{\mathsf{pub}(sk_1)}^{r_2}\}$ has an encryption cycle, while $\varphi_2 = \{x = \{\mathsf{pub}(sk_1)\}_{k_2}^{r_1},\ y = \{k_2\}_{\mathsf{pub}(sk_1)}^{r_2}\}$ does not.

A frame φ is *well-formed* if it satisfies the following conditions:

(i) φ is $\mathcal{R}$-reduced, that is, in normal form with respect to the rewriting system $\mathcal{R}$;
(ii) φ does not contain the symbols dec, pdec, sdec, $\mathsf{pdec_success}$, $\mathsf{sdec_success}$, fst, and snd;
(iii) terms in key position in φ are of the following forms, depending on their sort:
  - sorts *DKey* and *SKey*: names,
  - sort *EKey*: names and terms of the form $\mathsf{pub}(a)$,
  - sort *Data*: names and constants;
(iv) terms of type *Coins* may only be names, and appear as the third argument of an encryption; moreover, if such a name appears twice in φ then the encryption terms in which it appears are identical;
(v) φ has no encryption cycles;
(vi) for every subterm of φ of the form $\mathsf{enc}(T, k)$ where k is a name, T contains none of the constants w, c_0, c_1, ..., and T has no subterm of the form $\mathsf{enc}(S, 0)$ or $\mathsf{enc}(S, 1)$.

Condition (ii) indicates that we focus on the indistinguishability of expressions built from constructors; it does not preclude using other functions in the observations that may distinguish frames. Condition (iii) says that keys are atomic terms for symmetric encryptions, and terms of the form $\mathsf{pub}(a)$ for public-key encryptions. Similarly, condition (iv) says that coins are names and are used

only for encryptions, with different coins in each encryption. Condition (v) is the acyclicity requirement. Finally, condition (vi) restricts the occurrences of constants within plaintexts for deterministic encryption under strong keys (represented by names). For instance, this condition excludes the frame $\nu k.\{x_1 = \mathsf{enc}(\mathsf{c}_1, k),\, x_2 = \mathsf{enc}(\mathsf{c}_2, k)\}$, which is equivalent to $\nu a_1, a_2.\{x_1 = a_1,\, x_2 = a_2\}$ formally but not computationally if c_1 and c_2 happen to have the same bit-string implementations. More generally, when T_1 and T_2 are two terms such that $T_1 \neq_E T_2$, the encryptions $\mathsf{enc}(T_1, k)$ and $\mathsf{enc}(T_2, k)$ may behave like distinct fresh names formally but not computationally, unless the bit-string values of T_1 and T_2 collide with negligible probability.

We obtain:

Theorem 1 ($\approx_E$-soundness). *Let φ_1 and φ_2 be two well-formed frames such that $\varphi_1 \approx_E \varphi_2$. In any secure implementation, $[\![\varphi_1]\!] \approx [\![\varphi_2]\!]$.*

The proof of this theorem (in the extended version of this paper [1]) relies on a detailed formal analysis of static equivalence, and in particular on a decision procedure for the static equivalences under consideration. The theorem follows from a step-by-step complexity-theoretic validation of the decision procedure.

5 Application to Security Against Guessing Attacks

Weak secrets such as PINs and passwords sometimes serve as encryption keys. Their safe use is challenging because of the possibility of guessing attacks, in which data that depends on a weak secret allows an attacker to check guesses of the values of the weak secret. For example, if a message contains a fixed cleartext Hello, and it is encrypted under a password pwd drawn from a small dictionary, then an attacker that sees the message can try to decrypt it with all values in the dictionary until one yields the cleartext Hello, thus discovering a probable value for the password. The attacker may mount this attack off-line, avoiding detection. The attack is made possible by the fact that, given the data available to the attacker, pwd can be distinguished from another value pwd': $\mathsf{enc}_{\mathsf{string}}(\mathsf{Hello}, \mathsf{pwd}) \not\approx_E \mathsf{enc}_{\mathsf{string}}(\mathsf{Hello}, \mathsf{pwd'})$. Conversely, immunity to such guessing attacks can be formulated as a static equivalence between two frames, one that corresponds to what is actually available to the attacker and the other to a variant in which the weak secrets are replaced with fresh keys or with arbitrary other keys [17, 20].

We believe that, as suggested in the introduction, the treatment of guessing attacks in terms of static equivalence is attractive in several respects. This section shows that this treatment can be computationally sound. In comparison with the only previous computational justification for a formal criterion against guessing attacks [7], the present results have several strengths. First, they apply to a criterion formulated in terms of standard notions, rather than an ad hoc criterion. Consequently, they fit into a standard analysis method which can also deal with other properties and other kinds of attacks. In addition, they are more general, in that they immediately apply to scenarios with multiple weak secrets. Finally, it is satisfying that these results follow from theorems of somewhat broader interest.

In our formalism, modeling a password as a constant w of sort *Data*, we may say that the password is not revealed by a frame φ if $\varphi\{\mathsf{w} \mapsto \mathsf{c}_0\} \approx_E \varphi\{\mathsf{w} \mapsto \mathsf{c}_1\}$. The substitutions $\{\mathsf{w} \mapsto \mathsf{c}_0\}$ and $\{\mathsf{w} \mapsto \mathsf{c}_1\}$ correspond to instantiations of the password with distinct actual values; each of the frames represents what an attacker may obtain in the course of a protocol execution and then analyze off-line. The soundness of this formal notion is a corollary of Theorem 1, as is a generalization to multiple passwords. The formal notion can be applied to some examples from the literature (such as the EKE protocol [13, 7]), and the corollaries then yield computational guarantees for those examples.

Corollary 1 (Single password). *Assume a secure implementation. Let φ be a well-formed frame, let w be a constant of sort Data, and let c_0, c_1 be two fresh, distinct constants of sort Data. If $\varphi\{\mathsf{w} \mapsto \mathsf{c}_0\} \approx_E \varphi\{\mathsf{w} \mapsto \mathsf{c}_1\}$ then w is computationally hidden in φ: for all PPTIME-computable sequences of bit-strings κ_0, κ_1 with $\kappa_i(\eta) \in \{0,1\}^{\alpha_1(\eta)}$,*

$$[\![\varphi]\!]_{\eta, \mathsf{w} \mapsto \kappa_0(\eta)} \approx [\![\varphi]\!]_{\eta, \mathsf{w} \mapsto \kappa_1(\eta)}$$

Corollary 2 (Multiple passwords). *Assume a secure implementation. Let φ be a well-formed frame, let w_1, ..., w_n be n constants of sort Data, and let c_0, c_1, ..., c_n be $n+1$ fresh, distinct constants of sort Data. If $\varphi\{\mathsf{w}_1 \mapsto \mathsf{c}_1, \ldots, \mathsf{w}_n \mapsto c_n\} \approx_E \varphi\{\mathsf{w}_1 \mapsto \mathsf{c}_0, \ldots, \mathsf{w}_n \mapsto c_0\}$ then w_1, ..., w_n are computationally hidden in φ: for all (not necessarily pairwise distinct) PPTIME-computable sequences of bit-strings $\kappa_1 \ldots \kappa_n$, $\kappa'_1 \ldots \kappa'_n$ with $\kappa_i(\eta), \kappa'_i(\eta) \in \{0,1\}^{\alpha_1(\eta)}$,*

$$[\![\varphi]\!]_{\eta, \mathsf{w}_1 \mapsto \kappa_1(\eta), \ldots, \mathsf{w}_n \mapsto \kappa_n(\eta)} \approx [\![\varphi]\!]_{\eta, \mathsf{w}_1 \mapsto \kappa'_1(\eta), \ldots, \mathsf{w}_n \mapsto \kappa'_n(\eta)}$$

6 Conclusion

In this paper we investigate the computational foundations of a formal notion of data indistinguishability, static equivalence. We define a particular equational theory for which we can obtain a computational soundness result. Although they are largely based on ideas common in previous work, neither the equational theory nor our computational assumptions are straightforward. The main difficulties that we address relate to encryption under data values. Correspondingly, we obtain a soundness result for a formal criterion of protection against guessing attacks on those data values.

A direction for further work is the generalization of our results to other cryptographic primitives. For instance, certain password-based protocols make a sophisticated use of exponentiation, which we do not include in our equational theory. Yet other primitives, such as digital signatures, are important for trace properties and for process equivalences (more so than for static equivalences). We hope that, perhaps with these extensions, the present work may serve as a component of an eventual computational justification of process equivalences.

Acknowledgments. We thank Steve Kremer and the anonymous referees for helpful comments. This research was partly carried out while Mathieu Baudet was visiting the University of California at Santa Cruz and Bogdan Warinschi was at

Stanford University. It was partly supported by the National Science Foundation under Grants CCR-0204162, CCR-0208800, CCF-0524078, and ITR-0430594, and by the ARA SSIA Formacrypt and ACI Jeunes Chercheurs JC9005.

References

1. M. Abadi, M. Baudet, and B. Warinschi. Guessing attack and the computational soundness of static equivalence (extended version). Manuscript, 2006.
2. M. Abadi and V. Cortier. Deciding knowledge in security protocols under equational theories. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP'04)*, volume 3142 of *LNCS*, pages 46–58. Springer, 2004.
3. M. Abadi and V. Cortier. Deciding knowledge in security protocols under (many more) equational theories. In *Proc. 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 62–76, 2005.
4. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.
5. M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5(4):267–303, 1998.
6. M. Abadi and P. Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.
7. M. Abadi and B. Warinschi. Password-based encryption analyzed. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, volume 3580 of *LNCS*, pages 664–676. Springer, 2005.
8. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *Proc. 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 220–330, 2003.
9. G. Bana. *Soundness and completeness of formal logics of symmetric encryption.* PhD thesis, University of Pensilvania, 2004.
10. M. Baudet. Deciding security of protocols against off-line guessing attacks. In *Proc. 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 16–25, Alexandria, Virginia, USA, Nov. 2005.
11. M. Baudet, V. Cortier, and S. Kremer. Computationally sound implementations of equational theories against passive adversaries. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, volume 3580 of *LNCS*, pages 652–663. Springer, 2005.
12. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology – EUROCRYPT'00*, volume 1807 of *LNCS*, pages 139–155. Springer, 2000.
13. S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proc. 1992 IEEE Symposium on Security and Privacy (SSP'92)*, pages 72–84, 1992.
14. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *Proc. 20th IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 331–340, 2005.
15. M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. In *Proc. 14th IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 157–166, 1999.

16. V. Boyko, P. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *Advances in Cryptology – EUROCRYPT'00*, volume 1807 of *LNCS*, pages 156–171. Springer, 2000.
17. R. Corin, J. M. Doumen, and S. Etalle. Analysing password protocol security against off-line dictionary attacks. Technical report TR-CTIT-03-52, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, 2003.
18. R. Corin, S. Malladi, J. Alves-Foss, and S. Etalle. Guess what? Here is a new tool that finds some new guessing attacks (extended abstract). In *IFIP WG 1.7 and ACM SIGPLAN Workshop on Issues in the Theory of Security (WITS'03)*, pages 62–71, 2003.
19. S. Delaune and F. Jacquemard. A theory of dictionary attacks and its complexity. In *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 2–15, 2004.
20. C. Fournet. Private communication, 2002.
21. R. Gennaro and Y. Lindell. A framework for password-based authenticated key exchange. In *Advances in Cryptology – EUROCRYPT'03*, volume 2656 of *LNCS*, pages 524–543. Springer, 2003.
22. O. Goldreich and Y. Lindell. Session key generation using human passwords only. In *Advances in Cryptology – CRYPTO'01*, volume 2139 of *LNCS*, pages 403–432. Springer, 2001.
23. L. Gong. Verifiable-text attacks in cryptographic protocols. In *Proc. 9th IEEE Conference on Computer Communications (INFOCOM'90)*, pages 686–693, 1990.
24. L. Gong, T. M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, 1993.
25. J. Katz, R. Ostrovsky, and M. Yung. Practical password-authenticated key exchange provably secure under standard assumptions. In *Advances in Cryptology – EUROCRYPT'01*, volume 2045 of *LNCS*, pages 475–494. Springer, 2001.
26. P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proc. 2004 IEEE Symposium on Security and Privacy (SSP'04)*, pages 71–85, 2004.
27. G. Lowe. Analysing protocols subject to guessing attacks. *Journal of Computer Security*, 12(1):83–98, 2004.
28. D. Micciancio and B. Warinschi. Completeness theorems for the Abadi-Rogaway logic of encrypted expressions. *Journal of Computer Security*, 12(1):99–129, 2004.
29. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. Theory of Cryptography Conference (TCC'04)*, volume 2951 of *LNCS*, pages 133–151. Springer, 2004.
30. D. H. Phan and D. Pointcheval. About the security of ciphers (semantic security and pseudo-random permutations). In *Proc. Selected Areas in Cryptography (SAC'04)*, volume 3357 of *LNCS*, pages 185–200. Springer, 2004.
31. M. Steiner, G. Tsudik, and M. Waidner. Refinement and extension of encrypted key exchange. *ACM SIGOPS Oper. Syst. Rev.*, 29(3):22–30, 1995.
32. G. Tsudik and E. V. Herreweghen. Some remarks on protecting weak secrets and poorly-chosen keys from guessing attacks. In *Proc. 12th IEEE Symposium on Reliable Distributed Systems (SRDS'93)*, 1993.

Handling exp, $\times$ (and Timestamps) in Protocol Analysis*

Roberto Zunino and Pierpaolo Degano

Dipartimento di Informatica, Università di Pisa, Italy
{zunino, degano}@di.unipi.it

Abstract. We present a static analysis technique for the verification of cryptographic protocols, specified in a process calculus. Rather than assuming a specific, fixed set of cryptographic primitives, we only require them to be specified through a term rewriting system, with no restrictions. Examples are provided to support our analysis. First, we tackle forward secrecy for a Diffie-Hellman-based protocol involving exponentiation, multiplication and inversion. Then, a simplified version of Kerberos is analyzed, showing that its use of timestamps succeeds in preventing replay attacks.

1 Introduction

Process calculi [16] have been extensively used for cryptographic protocol specification and verification, exploiting formal methods. Several of these calculi (e.g. Spi [2]), however, use a specific set of cryptographic primitives, which is often entwined with the definition of the process syntax and semantics, e.g. by introducing pattern matching on encrypted messages. On the one hand, this simplifies the presentation of the calculus; also the verification tools only need to consider a given set of primitives. On the other hand, protocols using different primitives cannot be specified in the calculus as it is: one has to suitably extend it and to adapt existing tools to cope with the extensions. Of course, the new tools also needs new, adapted soundness proofs.

The applied pi calculus [1] instead does not fix the set of primitives. Its processes can exchange arbitrary terms, that are considered up to some equivalence relation. This relation can be defined by the user through an equational theory. In this scenario, adding primitives is done by adding the relevant equations to the theory, without changing the syntax of the processes. In other words, the applied pi effectively separates the semantics of the processes, which is fixed, from the semantics of the terms, which is user-defined.

In this paper, we present a technique for the static analysis of protocols specified in a (slight) variant of the applied pi. In our calculus, term equivalence is instead specified through an *arbitrary* rewriting system $\mathcal{R}$. Indeed, we do not put any restrictions on $\mathcal{R}$: it needs neither to be confluent nor terminating.

* Partly supported by the EU within the FETPI Global Computing, project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 413–427, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Our technique borrows from the control flow analysis (CFA) approach [19, 5] and from the algorithms for non-deterministic finite tree automata (NFTA) [10, 21]. As in the CFA, we extract a number of constraints from a protocol specification, expressed as a process. Then, we solve the constraints using the *completion* algorithm in [23], which turns out to be very similar to the one in [10, 21]. The result is a NFTA $\mathcal{F}$ describing a language which is an over-approximation of the set of terms exchanged by the protocol, in *all* their possible equivalent forms according to $\mathcal{R}$. Finally, the automaton $\mathcal{F}$ can be inspected to check a number of security properties of the protocol. Essentially, ours is a reachability analysis.

Exploiting this technique, we analyzed protocols using both standard cryptographic primitives, such as encryptions and signatures, as well as more "problematic" primitives such as exponentials and XOR. Exponentials are hard to deal with because their equational theory has many equations, and therefore equivalent terms may assume very different shapes. As a consequence, it is hard to find an accurate over-approximation for them. The literature often reports on studies carried out assuming only a few equations. For instance, from the web page of the AVISPA project [3] one sees examples with three equations for exp and inversion $(\bullet)^{-1}$ over finite-fields, analyzed through the tool in [6]. In the example of Sect. 4.1, we consider exp, $\times$, 1, and $(\bullet)^{-1}$, axiomatizing their interactions with twelve equations. Yet, our implementation of the presented analysis was able to prove the *forward secrecy* property for a protocol based on the Diffie-Hellman key exchange [8].

Our technique also offers a limited treatment of time. Here we report on the success of our tool for the verification of a simplified version of the Kerberos protocol [20, 18], involving timestamps. In our specification, we allow the disclosure of an old session key, mimicking a secret leak. The tool was able to prove the secrecy of messages exchanged in newer sessions, confirming the protocol is resilient of replay attacks with the compromised old key.

Finally, our technique allows for some composition of results. Albeit with some limitations, it is possible to analyze the components of a system independently, and then merge the results later to derive a sound analysis for the whole system.

A related approach to ours is in [4]. However, they only consider certain equational theories, e.g. without associativity, and define a semi-algorithm to obtain rewriting rules with "partial normal forms." They then use ProVerif to check processes equivalent, thus establishing security properties. Also, some decidability results for (a significant fragment of) the exponential theory are in [15]. Other applications of NFTA to security can be found in [12, 11]. There, protocols are specified through rewriting, rather than process calculi. Another interesting work is by Goubault [13], dealing with exponentials through rewriting. There, however, only exponentials with a fixed base are considered. Monniaux in [17] also uses NTFA for verifying protocols, when crypto primitives can be expressed through left-linear rewritings. Finally, there is an earlier analysis for the applied pi calculus in [22]. However, it only applies to free terms, subject to no rewriting.

Summary. In Sect. 2 we introduce background and notation. We present our calculus in Sect. 3, defining its dynamic semantics in Sect. 4. The same section

has the Diffie-Hellman example. Sect. 5 describes the static analysis and its application to Diffie-Hellman and Kerberos. In Sect. 6 we discuss compositionality.

2 Background and Notation

A non-deterministic finite tree automaton (NFTA) $\mathcal{A}$ is determined by its finite set of states $\mathcal{Q} = \{\texttt{@a}, \texttt{@b}, \ldots\}$ and its set of transitions. Transitions have the form $\texttt{@q} \to T$, where T is a generic term built using function symbols and states in $\mathcal{Q}$. For example, we consider the following $\mathcal{A}$:

$$\begin{array}{lll} \texttt{@a} \to 0 & \texttt{@a} \to 1 & \texttt{@a} \to 2 \\ \texttt{@b} \to \mathsf{nil} & \texttt{@b} \to \mathsf{cons}(\texttt{@a}, \texttt{@b}) & \texttt{@c} \to \mathsf{fst}(\texttt{@b}) \end{array}$$

In the above the function symbols are $0, 1, 2, \mathsf{nil}$ (nullary), fst (unary) and cons (binary). States $\mathcal{Q}$ are $\{\texttt{@a}, \texttt{@b}, \texttt{@c}\}$. Each state $\texttt{@q}$ has an associated language $[\texttt{@q}]_{\mathcal{A}}$, given by the set of the state-free terms reachable through transitions. For example, we have $\texttt{@b} \to \mathsf{cons}(\texttt{@a}, \texttt{@b}) \to \mathsf{cons}(\texttt{@a}, \mathsf{cons}(\texttt{@a}, \texttt{@b})) \to \mathsf{cons}(0, \mathsf{cons}(\texttt{@a}, \texttt{@b})) \to \mathsf{cons}(0, \mathsf{cons}(1, \texttt{@b})) \to \mathsf{cons}(0, \mathsf{cons}(1, \mathsf{nil})) = T$, and therefore $T \in [\texttt{@b}]_{\mathcal{A}}$.

A term rewriting system $\mathcal{R}$ is a set of rewriting rules, having the form $L \Rightarrow R$, where L, R are terms built using function symbols and variables. For example, the usual rewriting rules for pairs are:

$$\mathsf{fst}(\mathsf{cons}(X, Y)) \Rightarrow X \qquad \mathsf{snd}(\mathsf{cons}(X, Y)) \Rightarrow Y$$

In [23] an algorithm is described for computing the $\mathcal{R}$-*completion* of an automaton $\mathcal{A}$. The result is another automaton $\mathcal{F}$ such that its languages 1) include those of $\mathcal{A}$, and 2) are closed under rewriting. Formally, $\mathcal{F}$ is such that whenever $\texttt{@q} \to^*_{\mathcal{A}} \to^*_{\mathcal{R}} T$ also $\texttt{@q} \to^*_{\mathcal{F}} T$ for any $\texttt{@q}, T$. For instance, completing the $\mathcal{A}$ above, we obtain an $\mathcal{F}$ such that $\texttt{@c} \to^*_{\mathcal{F}} 1$. A very similar algorithm was presented in [10]. Once such an $\mathcal{F}$ is computed, it is possible to verify properties about the languages of $\mathcal{A}$ up-to rewriting by inspecting their over-approximations in $\mathcal{F}$.

For our purposes, we also want $\mathcal{F}$ to satisfy a set of *intersection constraints* $\mathcal{I}$, provided as an input to the algorithm. These constraints have the form $\texttt{@a} \cap \texttt{@b} \subseteq \texttt{@c}$, meaning that the intersection of the languages $[\texttt{@a}]_{\mathcal{F}}$ and $[\texttt{@b}]_{\mathcal{F}}$ must be included in $[\texttt{@c}]_{\mathcal{F}}$. The algorithm in [23] was adapted to handle $\mathcal{I}$ and is the basis for our analysis tool.

The time complexity of the completion algorithm is polynomial (assuming that the depth of each left hand side in any rewriting rule is constant).

3 Syntax

Our process calculus is a simplified version of the applied pi calculus [1], in that processes exchange values using a global public network channel. Values are simply represented as terms, up to the equivalence specified by a rewriting system $\mathcal{R}$. We write $\mathcal{T}$ for the set of terms. We also use $\mathcal{X}$ as a set of variables. The syntax of our calculus is rather standard.

$$\pi ::= \mathsf{in}\ x \mid \mathsf{out}\ M \mid [x = y] \mid \mathsf{let}\ x = M \mid \mathsf{new}\ x \mid \mathsf{repl} \mid \mathsf{chk}$$
$$P ::= \mathsf{nil} \mid \pi\ .\ P \mid (P|P)$$

We now briefly describe our calculus: its semantics will be given in Sect. 4. Intuitively, nil is a process that performs no actions; $\pi.P$ executes the prefix π and then behaves as P; $P_1|P_2$ runs concurrently the processes P_1 and P_2. Prefixes perform the following actions: $\mathsf{in}\ x$ reads a term from the network and binds x to it; $\mathsf{out}\ M$ sends a term to the network; $[x = y]$ compares the term bound to x and y and stops the process if they differ; $\mathsf{let}\ x = M$ simply locally binds x to the value of M; $\mathsf{new}\ x$ generates a fresh value and binds x to it; repl spawns an unlimited number of copies of the running process, which will run independently; chk is a special action that we use to model certain kinds of attacks, which we will address in Sect. 4.

Note that match $[x = y]$ is only allowed between variables. This is actually not a restriction, since matching between arbitrary terms, e.g. $[M = N].P$ can be expressed by $\mathsf{let}\ x = M\ .\mathsf{let}\ y = N\ .[x = y].P$.

As usual, the *bound* variables in a process are those under a let,new, or in prefix; the others are *free*. A process with no free variables is *closed*.

Given a process, we use addresses $\theta \in \{\mathsf{n}, \mathsf{l}, \mathsf{r}\}^*$ to point to its subprocesses. Intuitively, n chooses the continuation P for a process $\pi.P$, while l and r choose the left and right branch of a parallel $P_1|P_2$, respectively. An address θ is a concatenation of these selectors, singling out the subprocess $P@\theta$ as defined below. We write ϵ for the empty string.

$$\begin{array}{ll} P@\epsilon = P & (P_1|P_2)@\theta\mathsf{l} = P_1@\theta \\ \pi.P@\theta\mathsf{n} = P@\theta & (P_1|P_2)@\theta\mathsf{r} = P_2@\theta \end{array}$$

4 Dynamic Semantics

Given a closed process P, we define its semantics through a multiset rewriting system [14, 7]. A state is a multiset σ of parallel threads. Each thread is formed by an environment $\rho \in \mathcal{X} \to \mathcal{T}$ and a continuation address θ singling out a subprocess of P. We write such a thread as $\langle \rho, \theta \rangle$. Intuitively, $\langle \rho, \theta \rangle$ runs the process $P@\theta$ under the bindings in ρ. The initial state is $\langle \emptyset, \epsilon \rangle$.

We extend ρ homomorphically to terms: $\rho(M)$ replaces variables in M with the value they are bound to in ρ. Also, as a handy convention, if $P@\theta = P_1|P_2$, we write $\langle \rho, \theta \rangle$ for the multiset $\{\langle \rho, \mathsf{l}\theta \rangle, \langle \rho, \mathsf{r}\theta \rangle\}$, or its further expansion, so that threads in the state never have continuation addresses θ' such that $P@\theta'$ has the form $P_1|P_2$.

Our semantics is given by the rules in Fig. 1. Local rules only care about one or two elements of the current state: these elements are rewritten independently of the rest of the state, which does not change. All the rules fire a prefix, advancing the current continuation address θ to $\mathsf{n}\theta$, except for rule Rew.

Rule Comm performs communication between threads. Rule Out outputs a term to the external environment. Since $\mathsf{out}\ M$ may be handled by either Comm or Out, there is no guarantee that outputs have a corresponding input; instead,

Local Rules

$$\text{Comm}\ \frac{P@\theta_1 = \mathsf{in}\ x\ .P' \qquad P@\theta_2 = \mathsf{out}\ M\ .P'' \qquad \rho_1' = \rho_1[x \mapsto \rho_2(M)]}{\langle \rho_1, \theta_1 \rangle, \langle \rho_2, \theta_2 \rangle \xrightarrow{\mathsf{comm}\ \theta_1, \theta_2, \rho_2(M)} \langle \rho_1', \mathsf{n}\theta_1 \rangle, \langle \rho_2, \mathsf{n}\theta_2 \rangle}$$

$$\text{Out}\ \frac{P@\theta = \mathsf{out}\ M\ .P'}{\langle \rho, \theta \rangle \xrightarrow{\mathsf{out}\ \theta, \rho(M)} \langle \rho, \mathsf{n}\theta \rangle}$$

$$\text{Match}\ \frac{P@\theta = [x = y].P' \quad \rho(x) = \rho(y)}{\langle \rho, \theta \rangle \xrightarrow{\tau} \langle \rho, \mathsf{n}\theta \rangle} \qquad \text{Let}\ \frac{P@\theta = \mathsf{let}\ x = M\ .P'}{\langle \rho, \theta \rangle \xrightarrow{\tau} \langle \rho[x \mapsto \rho(M)], \mathsf{n}\theta \rangle}$$

$$\text{Repl}\ \frac{P@\theta = \mathsf{repl}.P'}{\langle \rho, \theta \rangle \xrightarrow{\tau} \langle \rho, \theta \rangle, \langle \rho, \mathsf{n}\theta \rangle} \qquad \text{Rew}\ \frac{\rho(x) \rightarrow_{\mathcal{R}} M}{\langle \rho, \theta \rangle \xrightarrow{\tau} \langle \rho[x \mapsto M], \theta \rangle}$$

Global Rules

$$\text{New}\ \frac{P@\theta = \mathsf{new}\ x\ .P' \quad \hat{x} = \mathsf{genFresh}()}{\sigma, \langle \rho, \theta \rangle \xrightarrow{\tau} \sigma, \langle \rho[x \mapsto \hat{x}], \mathsf{n}\theta \rangle} \qquad \text{Chk}\ \frac{P@\theta = \mathsf{chk}.P'}{\sigma, \langle \rho, \theta \rangle \xrightarrow{\mathsf{chk}} \langle \rho, \mathsf{n}\theta \rangle}$$

Fig. 1. Multiset Rewriting Rules

they may simply cause a *barb*, i.e. an action observed only by the external environment. Note that there is no rule for input, and therefore processes can never receive a value from the environment – for studying security issues our processes will explicitly contain an adversary.

Rule Match, allows a process to continue only if x and y are bound to the same term. Rule Let simply updates ρ with the new binding. Rule Repl allows for spawning a new copy of P'. In Rew, the thread rewrites the term bound by x, thus performing an internal computation step; note that these internal steps may lead a matching to succeed.

Global rules instead look at the whole state. Rule New is not completely standard, and it generates a *fresh* value $\hat{x}$ for the variable x. Here we postulate that 1) a constant (nullary function) symbol $\bar{x}$ exists for each variable bound by new in P, and 2) two function symbols val, next exist, subject to no rewriting in $\mathcal{R}$. Note that we only need a finite number of such $\bar{x}$, since there are only finitely many variables in P and we have *no* α-conversion. When rule New is applied, the $\hat{x}$ is generated by a genFresh() primitive, which we assume to choose among $\mathsf{val}(\bar{x}), \mathsf{val}(\mathsf{next}(\bar{x})), \mathsf{val}(\mathsf{next}(\mathsf{next}(\bar{x}))), \ldots$. To make it possible to track new-generated values to their new prefix in P, we require that all new-bound variables are distinct, and therefore so are their related constants $\bar{x}$. Note that this representation prevents an adversary *Adv* to deduce any instance of $\hat{x}$ from other instances he knows, even if *Adv* can use val and next.

Rule Chk is peculiar: when a chk prefix is fired *all* the other threads are aborted, and the thread continues its execution alone. For simplicity, we admit only one firing of chk. We use this special prefix to model some kind of attacks. For instance, suppose we want to study the case in which the adversary learns some secret term S, maybe by corrupting some participant to the protocol. A straightforward way to model this attack would be simply adding out S to the

protocol, disclosing S. While this would work, in many cases giving this kind of power to the adversary might allow for trivial attacks. Instead, to keep the game fair, we could restrict the interaction between the adversary and the participants after the disclosure of S. For example, we could imagine that it would take a long time for the adversary to obtain S, and meanwhile the participants have terminated the protocol run, either normally or because of a time-out.

A possible usage of chk is the following. The adversary, after having learnt S, is only allowed to run alone, and possibly use this new knowledge to decrypt messages it learnt in the past. In our calculus, we model this scenario as

$$(Proto|Adv)|\mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ S\ .\mathsf{nil}|Adv)$$

Usually, the process Adv is chosen independently of the protocol, modeling the capabilities of any adversary, as we shall do in our examples.

Note that we include the adversary process twice. First, the adversary can interact with the protocol. Later, when chk is fired, the adversary can learn S and go on with its computation, without being able to communicate with the protocol participants. Since we want to allow the adversary to keep its knowledge across the chk firing, we simply save it in the variable *know* before the chk, and make it again available to the adversary later on. Note that, while *know* is only a single term, it can be a cons-list of all the terms known by the adversary. Therefore *know* actually can bring all the old adversary knowledge into the new world, provided we have a primitive for pairing. In the next section, we show such a use of chk.

Another interesting use of chk we found is for modeling *timestamps*, as we will show in Sect. 5.3.

4.1 Diffie-Hellman Example

We consider the following key-exchange protocol, based on Diffie–Hellman [8].

1. $A \to \mathsf{all} : \mathsf{g}$	4. $A \to B : \{\mathsf{m}\}_{\mathsf{g}^{\mathsf{ab}}}$
2. $A \to B : \{\mathsf{g}^{\mathsf{a}}\}_{\mathsf{k1}}$	5. ...
3. $B \to A : \{\mathsf{g}^{\mathsf{b}}\}_{\mathsf{k2}}$	6. $A \to \mathsf{all} : \mathsf{k1}, \mathsf{k2}$

Initially, the principals A and B share two long term secret keys $\mathsf{k1}, \mathsf{k2}$, and agree on a public finite field $\mathsf{GF}[p]$ (where p is a large prime), and public generator g of $\mathsf{GF}[p]^*$. In the second step, principal A generates a nonce a and sends B the result of $\mathsf{g}^{\mathsf{a}} (\bmod p)$, encrypted with the key $\mathsf{k1}$. In the third step, B does the same, with its own nonce b and key $\mathsf{k2}$. Since both principals know the long term keys, they can compute $(\mathsf{g}^{\mathsf{b}})^{\mathsf{a}} = \mathsf{g}^{\mathsf{ab}} = (\mathsf{g}^{\mathsf{a}})^{\mathsf{b}} \pmod p$ and use this value as a session key to exchange the message m in the fourth step.

We study the robustness of this protocol against the active Dolev–Yao [9] adversary (such an adversary has full control over the public network, can reroute, discard or forge messages; further, he can apply any algebraic operation to terms learnt before). The adversary we use runs all the available operations in a non-deterministic way. Doing this, its behaviour encompasses that of any arbitrary Dolev-Yao adversary.

More in detail, we are interested in the *forward secrecy* of the message m. That is, we want m to be kept secret even though later on the long term keys k1, k2 are disclosed (last step).

We define the algebra by adapting the rewriting rules for encryption, multiplication, exponentiation, and inversion from [15]:

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{dec}(\mathsf{enc}(X,K),K) \Rightarrow X & \\ \mathsf{fst}(\mathsf{cons}(X,Y)) \Rightarrow X & \mathsf{snd}(\mathsf{cons}(X,Y)) \Rightarrow Y \\ \times(1,X) \Rightarrow X & \mathsf{exp}(\mathsf{inv}(X),Y) \Rightarrow \mathsf{inv}(\mathsf{exp}(X,Y)) \\ \times(X,Y) \Rightarrow \times(Y,X) & \times(X,\times(Y,Z)) \Rightarrow \times(\times(X,Y),Z) \\ \mathsf{exp}(X,1) \Rightarrow X & \mathsf{exp}(1,X) \Rightarrow 1 \\ \mathsf{inv}(\mathsf{inv}(X)) \Rightarrow X & \mathsf{exp}(\mathsf{exp}(X,Y),Z) \Rightarrow \mathsf{exp}(X,\times(Y,Z)) \\ \mathsf{inv}(1) \Rightarrow 1 & \mathsf{exp}(\times(Y,Z),X) \Rightarrow \times(\mathsf{exp}(Y,X),\mathsf{exp}(Z,X)) \\ \times(X,\mathsf{inv}(X)) \Rightarrow 1 & \mathsf{inv}(\times(X,Y)) \Rightarrow \times(\mathsf{inv}(X),\mathsf{inv}(Y)) \end{array} \right\}$$

Note that the algebra $\mathcal{A}$ defined by the above rewriting rules is not the same algebra of $\mathsf{GF}[p]^*$. In fact, $\mathcal{A}$ satisfies more equations than the ones that hold in $\mathsf{GF}[p]^*$. For instance, operations in $\mathcal{A}$ do not specify which modulus is being used; e.g., inversion modulo n is simply written as $\mathsf{inv}(X)$ rather than $\mathsf{inv}(X,n)$. Therefore, we have *(a)* $\times(X,\mathsf{inv}(X)) = 1$ and *(b)* $\mathsf{exp}(Y,\times(X,\mathsf{inv}(X))) = Y$. However, *(a)* holds in $\mathsf{GF}[p]^*$ only if $\mathsf{inv}(X)$ is performed modulo p, while *(b)* holds only if $\mathsf{inv}(X)$ is performed modulo $\varphi(p) = p - 1$ (where φ is the Euler function). In spite of $\mathcal{A}$ being not equal to the $\mathsf{GF}[p]^*$ algebra, the equations that hold in $\mathsf{GF}[p]^*$ do hold in $\mathcal{A}$.

We use the following process:

$$\begin{aligned} P =& \mathsf{new}\ g\ .(DY|\mathsf{new}\ k1\ .\mathsf{new}\ k2\ .(Proto|Chk)) \\ Chk =& \mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ k1\ .\mathsf{out}\ k2\ .\mathsf{nil}|DY)) \\ Proto =& A|B \\ A =& \mathsf{repl}.\mathsf{new}\ a\ .\mathsf{out}\ \mathsf{enc}(\mathsf{exp}(g,a),k1)\ .\mathsf{in}\ x\ . \\ & \quad \mathsf{let}\ k = \mathsf{exp}(\mathsf{dec}(x,k2),a)\ .\mathsf{out}\ \mathsf{enc}(\mathsf{m},k)\ .\mathsf{nil} \\ B =& \mathsf{repl}.\mathsf{new}\ b\ .\mathsf{in}\ x\ .\mathsf{out}\ \mathsf{enc}(\mathsf{exp}(g,b),k2)\ . \\ & \quad \mathsf{let}\ k = \mathsf{exp}(\mathsf{dec}(x,k1),b)\ .\mathsf{in}\ n\ .\mathsf{out}\ \mathsf{hash}(\mathsf{dec}(n,k))\ .\mathsf{nil} \end{aligned}$$

$$\begin{aligned} DY =& \mathsf{repl}.((\mathsf{new}\ nonce\ .\mathsf{out}\ nonce\ .\mathsf{nil}|\mathsf{out}\ g\ .\mathsf{out}\ 1\ .\mathsf{nil})| \\ & \mathsf{in}\ x\ .\mathsf{in}\ y\ .\mathsf{out}\ \mathsf{enc}(x,y)\ .\mathsf{out}\ \mathsf{dec}(x,y)\ .\mathsf{out}\ \mathsf{exp}(x,y)\ . \\ & \mathsf{out}\ \times(x,y)\ .\mathsf{out}\ \mathsf{inv}(x)\ .\mathsf{out}\ \mathsf{cons}(x,y)\ .\mathsf{out}\ \mathsf{fst}(x)\ .\mathsf{out}\ \mathsf{snd}(x)\ . \\ & \mathsf{out}\ \mathsf{hash}(x)\ .\mathsf{out}\ \mathsf{val}(x)\ .\mathsf{out}\ \mathsf{next}(x)\ .\mathsf{nil}) \end{aligned}$$

The specification P combines the protocol participants with the DY adversary. The principal B outputs the hash of the exchanged message, just as a witness. We also add a Chk process for the explicit disclosure of the secret keys: of course, this only happens after the chk prefix is fired.

We expect P to ensure the secrecy of the message m. Further, we expect this secrecy property to still hold even after the chk fires and thus the long term keys $k1, k2$ are disclosed. This is the *forward secrecy* property.

5 Static Semantics

Our analysis over-approximates the values that processes exchange at run-time. These sets of values result from solving a set of constraints generated from a given process P.

We decided to represent these sets of values as the languages associates with the states of a finite tree automaton. Some of the constraints extracted from P can be expressed as transitions (e.g. $\{\mathsf{f}(\mathsf{g}(x), y) | x \in X \wedge y \in Y\} \subseteq Z$ becomes $\texttt{@z} \to \mathsf{f}(\mathsf{g}(\texttt{@x}), \texttt{@y})$), forming an automaton $\mathcal{A}$. The others are intersection constraints, and form a set $\mathcal{I}$, with typical element $\texttt{@a} \cap \texttt{@b} \subseteq \texttt{@c}$. Of course, we also require our sets of values be closed under rewritings in $\mathcal{R}$.

Our tool, supplied with $\mathcal{A}, \mathcal{I}, \mathcal{R}$, computes an automaton $\mathcal{F}$ such that its languages include those of $\mathcal{A}$, satisfy $\mathcal{I}$, and are closed under $\mathcal{R}$. Once done that, we can check a number of properties about P by simply inspecting $\mathcal{F}$.

We first give some intuition behind the construction of $\mathcal{A}$ and $\mathcal{I}$. Roughly, we follow the data-flow between processes depicted in Fig. 2. In the figure, the arrows towards/from processes represent inputs and outputs, respectively, while bullets represent the data-flow points which we focus on in our analysis. For each bullet, we compute an approximation for the set of values that flow through it.

More in detail, we generate a dedicated state of $\mathcal{A}$ for each bullet, and add transitions between states following the arrows in the figure. Formally, the states of $\mathcal{A}$ are:

- `@in`, `@out`, `@chk-in`, `@chk-out`;
- `@in-`$b\theta$ and `@out-`$b\theta$, for each θ such that $P@\theta = P_1 | P_2$, and $b \in \{\mathsf{l}, \mathsf{r}\}$;
- `@in-n`θ and `@out-n`θ, for each θ such that $P@\theta = \mathsf{repl}.P'$;
- `@inters-`θ, for each θ such that $P@\theta = [x = y].P'$;
- `@x` and `@x-val`, for each $\mathsf{new}\ x$ occurring in P;
- `@x`, for each $\mathsf{let}\ x = M$ and $\mathsf{in}\ x$ occurring in P.

We generate the transitions of the automaton $\mathcal{A}$ and the intersection constraints $\mathcal{I}$ using the *gen* function, recursively defined in Fig. 3. The expression *gen*

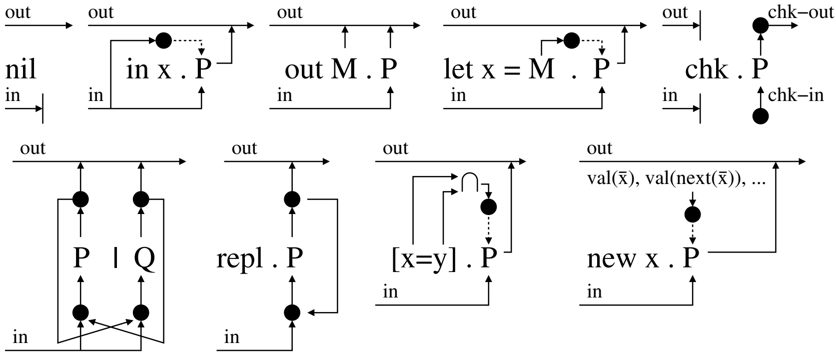


Fig. 2. Data Flow

$$
\begin{aligned}
&gen(\theta, \mathsf{nil}, \zeta, in, out) = \emptyset\\
&gen(\theta, \mathsf{in}\ x\ .P, \zeta, in, out) = (\mathtt{@x} \to in), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \mathtt{@x}], in, out)\\
&gen(\theta, \mathsf{out}\ M\ .P, \zeta, in, out) = (out \to \zeta(M)), gen(\mathsf{n}\theta, P, \zeta, in, out)\\
&gen(\theta, (P|Q), \zeta, in, out) = (out \to \mathtt{@out\text{-}l}\theta), (out \to \mathtt{@out\text{-}r}\theta),\\
&\quad (\mathtt{@in\text{-}l}\theta \to in), (\mathtt{@in\text{-}l}\theta \to \mathtt{@out\text{-}r}\theta), (\mathtt{@in\text{-}r}\theta \to in), (\mathtt{@in\text{-}r}\theta \to \mathtt{@out\text{-}l}\theta),\\
&\quad gen(\mathsf{l}\theta, P, \zeta, \mathtt{@in\text{-}l}\theta, \mathtt{@out\text{-}l}\theta), gen(\mathsf{r}\theta, Q, \zeta, \mathtt{@in\text{-}r}\theta, \mathtt{@out\text{-}r}\theta)\\
&gen(\theta, \mathsf{repl}.P, \zeta, in, out) = (out \to \mathtt{@out\text{-}n}\theta), (\mathtt{@in\text{-}n}\theta \to in),\\
&\quad (\mathtt{@in\text{-}n}\theta \to \mathtt{@out\text{-}n}\theta), gen(\mathsf{n}\theta, P, \zeta, \mathtt{@in\text{-}n}\theta, \mathtt{@out\text{-}n}\theta)\\
&gen(\theta, \mathsf{let}\ x = M\ .P, \zeta, in, out) = (\mathtt{@x} \to \zeta(M)), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \mathtt{@x}], in, out)\\
&gen(\theta, \mathsf{new}\ x\ .P, \zeta, in, out) = (\mathtt{@x} \to \mathsf{val}(\mathtt{@x\text{-}val})),\\
&\quad (\mathtt{@x\text{-}val} \to \bar{x}), (\mathtt{@x\text{-}val} \to \mathsf{next}(\mathtt{@x\text{-}val})), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \mathtt{@x}], in, out)\\
&gen(\theta, [x = y].P, \zeta, in, out) = (\zeta(x) \cap \zeta(y) \subseteq \mathtt{@inters\text{-}}\theta),\\
&\quad gen(\mathsf{n}\theta, P, \zeta[x, y \mapsto \mathtt{@inters\text{-}}\theta], in, out)\\
&gen(\theta, \mathsf{chk}.P, \zeta, in, out) = gen(\mathsf{n}\theta, P, \zeta, \mathtt{@chk\text{-}in}, \mathtt{@chk\text{-}out})
\end{aligned}
$$

Fig. 3. Extraction of $\mathcal{A}, \mathcal{I}$ from a process

$(\theta, P', \zeta, in, out)$ generates the transitions and intersection constraints for $P' = P@\theta$, a subprocess of P[1]. The *static environment* $\zeta \in \mathcal{X} \to \mathcal{Q}$ keeps track of which state of $\mathcal{A}$ is used to approximate the sets of values that can be dynamically bound to each variable in scope. The *in* and *out* parameters define the states for the approximation of the values that can be received and sent by P', respectively. Initially, *gen* is called as $gen(\epsilon, P, \emptyset, \mathtt{@in}, \mathtt{@out})$ to generate $\mathcal{A}, \mathcal{I}$ for the whole process P.

No productions are generated for nil. For in $x\ .P'$, we generate a new state @x, and a transition from it to *in* to include inputs in its language. Then, we update ζ (the dotted line in Fig. 2) by binding x to @x, and proceed recursively with the continuation P'. Outputs as out $M\ .P'$ generate a transition from the *out* state to $\zeta(M)$, the term obtained by replacing all the variables in M with their corresponding states; we then proceed recursively for P'. For example, the generated transitions for $P = \mathsf{in}\ x\ .\mathsf{out}\ \mathsf{f}(x)\ .\mathsf{out}\ \mathsf{g}(x)\ .\mathsf{nil}$ are $\mathtt{@x} \to \mathtt{@in}, \mathtt{@out} \to \mathsf{f}(\mathtt{@x}), \mathtt{@out} \to \mathsf{g}(\mathtt{@x})$. Note that each output *contributes* to the language of @out by adding transitions to those already generated. This is depicted in Fig. 2 by the *out* arrow going straight from left to the right and collecting possible outputs from below. As seen in the figure, this happens for all processes, except for chk.

Parallel processes such as $P|Q$ are handled by creating four dedicated states for input and output of the left and right branch, then adding transitions to cross-connect inputs and outputs as in Fig. 2. Replication repl.P is done in a similar fashion, with a loopback transition.

For let bindings, we simply create a new state for the approximation of the bound value, and update ζ accordingly. A new $x\ .P'$ causes the generation of

[1] The parameter P' of *gen* is actually redundant since it is determined by θ, but its presence allows for a simple definition.

transitions for the language $\mathsf{val}(\bar{x}), \mathsf{val}(\mathsf{next}(\bar{x})), \mathsf{val}(\mathsf{next}(\mathsf{next}(\bar{x}))), \ldots$ using the two states `@x` and `@x-val`; then, we update ζ to bind x to this language.

A match $[x = y].P'$ creates a new state `@inters-`θ for the (approximation of the) intersection of values hold by x and y, together with the associated intersection constraint; in the analysis of P' we use this new state for both $\zeta(x)$ and $\zeta(y)$.

When a chk is fired, the continuation runs in an isolated world, therefore in the analysis we simply reset in, out to new independent states and proceed recursively. Note that ζ is not changed, and that bound variables bring their values into the new world (e.g. in x .chk.out x .nil).

Note that our analysis generates no transitions for states `@in` and `@chk-in`: their language is therefore empty. In fact, top-level processes receive no value from their environment; this reflects the absence of an input rule.

Matching and Precision. Consider the following process:

$$P_1 = \mathsf{out\ cons}(0,0)\ .\mathsf{out\ cons}(1,1)\ .\mathsf{nil} \mid$$
$$\mathsf{in}\ x\ .\mathsf{let}\ f = \mathsf{fst}(x)\ .\mathsf{let}\ z = 0\ .[f = z].\mathsf{out\ snd}(x)\ .\mathsf{nil}$$

At run-time, the last out $\mathsf{snd}(x)$ can output 0, only. However, our analysis of the match $[f = z]$ does only refine the approximation of f and z, and not that of x. Therefore, in the analysis, a single state is used for the values of x before and after the match. The result of the analysis is that the last out may output either 0 or 1.

A more precise result can be obtained by using instead the following pattern:

$$P_2 = \mathsf{out\ cons}(0,0)\ .\mathsf{out\ cons}(1,1)\ .\mathsf{nil} \mid$$
$$\mathsf{in}\ x\ .\mathsf{let}\ y = \mathsf{cons}(0, \mathsf{snd}(x))\ .[x = y].\mathsf{out\ snd}(x)\ .\mathsf{nil}$$

Here the analysis of the match refines the approximation of x itself, and therefore deduces that the last output can only be 0. We will use this style of matching in our examples.

5.1 Subject Reduction

Here, we establish the soundness for our analysis.

First, we define an address compatibility relation $\sim$ over addresses. Roughly speaking, $\theta_1 \sim \theta_2$ means that at run-time a thread running $P@\theta_1$ could communicate with a thread running $P@\theta_2$. The actual $\sim$ over-approximates run-time communication, and simply checks if the two addresses point to processes either at different branches of the same parallel, or under the same replication. We also take into account the presence of the chk prefix, since its continuation cannot interact with previously spawned threads.

More in detail, we say that chk *occurs* between θ and $\theta'\theta$ iff for some θ_a, θ_b we have $\theta' = \theta_a\theta_b$ and $P@\theta_b\theta = \mathsf{chk}.P'$. Then, the address compatibility relation $\sim$ is the minimum relation such that

- if chk does neither occur between $\mathsf{l}\theta$ and $\theta_l\mathsf{l}\theta$, nor between $\mathsf{r}\theta$ and $\theta_r\mathsf{r}\theta$, then $\theta_l\mathsf{l}\theta \sim \theta_r\mathsf{r}\theta$
- if $P@\theta = \mathsf{repl}.P'$ and chk does neither occur between θ and $\theta_1\theta$, nor between θ and $\theta_2\theta$, then $\theta_1\theta \sim \theta_2\theta$

The following lemma ensures that the relation $\sim$ actually encompasses all run-time communications. Its proof can be done by induction on the number of computation steps.

Lemma 1. *If* $P \rightarrow^* \xrightarrow{\mathsf{comm}\ \theta_1,\theta_2,M}$, *then* $\theta_1 \sim \theta_2$.

Input and output states related to compatible addresses satisfy the following inclusion property. The proof of this lemma is by structural induction on P.

Lemma 2. *If* $\theta_1 \sim \theta_2$, *then we have* $[\texttt{@out-}\theta_1]_{\mathcal{F}} \subseteq [\texttt{@in-}\theta_2]_{\mathcal{F}}$, *provided these states exist.*

The following theorem ensures that our analysis is sound, relating the dynamic semantics to the static one.

Theorem 1 (Subject Reduction). *Given* P, *let* $\mathcal{F}$ *be the automata resulting from the analysis. Assume* $\langle \emptyset, \epsilon \rangle \rightarrow^* \xrightarrow{\alpha} \sigma, \langle \rho, \theta \rangle$.

1. $\forall x \in \mathsf{dom}(\rho).\ \rho(x) \in [\texttt{@x}]_{\mathcal{F}}$
2. *if* $\alpha = (\mathsf{out}\ \theta\ , M)$ *and* chk *was not fired before* α, *then* $M \in [\texttt{@out}]_{\mathcal{F}}$
3. *if* $\alpha = (\mathsf{out}\ \theta\ , M)$ *and* chk *was fired before* α, *then* $M \in [\texttt{@chk-out}]_{\mathcal{F}}$

Proof (Sketch). By induction on the number of computation steps. First, we consider property (1): for this, we only need to check the rules that update the environment ρ.

When the Comm rule is applied, yielding to comm $\theta_1, \theta_2, \rho_2(M)$, by Lemma 1 we have $\theta_1 \sim \theta_2$. We look for the transitions for in x and out M generated by *gen*(). These transitions have the form $\texttt{@x} \rightarrow in$ and $out \rightarrow \zeta(M)$, where $in = \texttt{@in-}\theta_i$ and $out = \texttt{@out-}\theta_o$. The addresses θ_i, θ_o, in general, are not the same as θ_1, θ_2, but they are strictly related so that we have also $\theta_i \sim \theta_o$. By Lemma 2, $\rho_1'(x) = \rho_2(M) \in [\texttt{@out-}\theta_o]_{\mathcal{F}} \subseteq [\texttt{@in-}\theta_i]_{\mathcal{F}} \subseteq [\texttt{@x}]_{\mathcal{F}}$, provided that $\zeta(M)$ is a correct approximation of $\rho_2(M)$. For this last proof obligation, we note that when there is no match involving variables in M, we have $\zeta(x) = \texttt{@x}$, so inductive hypothesis and structural induction on M suffices. Otherwise, if there is a match, we have $\zeta(x) = \texttt{@inters-}\theta_m$ for some x occurring in M. Here we first proceed by structural induction on P, obtaining $\rho_2(x) \in [\texttt{@inters-}\theta_m]_{\mathcal{F}}$, and then continue as for the no-match case. This shows that property (1) is preserved by Comm.

We now tackle property (1) for the other rules. The Let case is straightforward: we generated the transition $\texttt{@x} \rightarrow \zeta(M)$, so we have $\rho(M) \in [\texttt{@x}]_{\mathcal{F}}$. Rule New also poses no problem, because the fresh term returned by genFresh() is chosen among the terms in the language of @x. Finally, environment updates by rule Rew are harmless, the languages of $\mathcal{F}$ being closed under rewritings.

For properties (2,3), only rule Out may cause out θ, M. Here, structural induction on P is sufficient to show that $M \in [\texttt{@out-}\theta']_{\mathcal{F}}$, where $P@\theta'$ ranges from $P@\theta$ to i) the enclosing chk.P', if any, or otherwise to ii) the top level P. From this, we deduce $M \in [\texttt{@out}]_{\mathcal{F}}$ or $M \in [\texttt{@chk-out}]_{\mathcal{F}}$, depending on whether ii) or i) applies, respectively. □

5.2 Diffie-Hellman Example (Continued)

We ran the above analysis on the protocol specified in Sect. 4.1, computing the result $\mathcal{F}$. Our tool generated an $\mathcal{F}$ having 47 states and 865 transitions. Our analysis was able to establish forward secrecy, as m $\notin [\texttt{@out-chk}]_{\mathcal{F}}$.

5.3 Kerberos

We now study a protocol involving timestamps. We chose a simplified version of Kerberos [20, 18].

In this protocol, a key exchange is performed by an authentication server AS, a client C and a server S. Initially, the authentication server shares long term keys with the client (kc) and with the server (ks). Upon request from the client, AS generates a fresh key kcs and sends it to the client encrypted with kc. Further, AS also provides a certificate for the freshness of kcs, made of the kcs key itself and the current time, both encrypted by ks. The server S can decrypt the certificate and ensure that kcs is indeed fresh by checking the timestamp. After that, C and S use kcs to exchange a session key $ksess$, and then proceed exchanging messages encrypted with $ksess$.

We study the rôle of timestamps in the protocol. To that purpose, we introduce a vulnerability in the server S. In our implementation, we let the server to disclose kcs, potentially mining the security of the protocol. However, to keep the game fair, disclosure may only happen after a long time since the timestamp for kcs has been generated. We model this through the occurrence of a chk. Hopefully, if timestamps are properly checked, disclosing a old kcs will not disrupt new sessions of the protocol.

In our specification, we abstract the actual timestamps values with two constants before and after. Initially, the protocol uses only before: any other timestamp value is considered not valid, being in the far past or far future. After chk, the before timestamp has expired, and the protocol has moved to newer timestamps, represented by after. Similarly, we use msg1 and msg2 for the messages exchanged by C and S before and after the chk, respectively.

We expect this faulty protocol implementation not to disclose msg1 until a chk occurs. After chk, we do expect msg1 to be disclosed, but we hope any new msg2 messages to be kept secret.

We specify the above as follows: (we omit parentheses in $P_1|\cdots|P_n$ for readability)

$$P = DY|\mathsf{new}\ kc\ .\mathsf{new}\ ks\ .(AS|C|S)$$

$$AS = \mathsf{repl.new}\ kcs\ .\mathsf{in}\ nonce\ .\mathsf{out}\ \mathsf{enc}(\mathsf{cons}(nonce, kcs), kc)\ .$$
$$\mathsf{out}\ \mathsf{enc}(\mathsf{cons}(kcs, \mathsf{before}), ks)\ .\mathsf{nil}$$

$$\begin{aligned}
C =& \mathsf{repl.new}\ nonce\ .\mathsf{out}\ nonce\ .\mathsf{in}\ ticket\ .\mathsf{in}\ cert\ .\\
& \mathsf{let}\ ticketCorrect = \mathsf{enc}(\mathsf{cons}(nonce, \mathsf{snd}(\mathsf{dec}(ticket, kc))), kc)\ .\\
& [ticket = ticketCorrect].\mathsf{let}\ kcs = \mathsf{snd}(\mathsf{dec}(ticket, kc))\ .\\
& \mathsf{new}\ ksess\ .\mathsf{out}\ \mathsf{enc}(ksess, kcs)\ .\mathsf{out}\ cert\ .\mathsf{out}\ \mathsf{enc}(\mathsf{msg1}, ksess)\ .\mathsf{nil}\\
S =& \mathsf{repl.in}\ tsess\ .\mathsf{in}\ cert\ .\mathsf{let}\ sess = \mathsf{dec}(cert, ks)\ .\\
& \mathsf{let}\ sessCorrect = \mathsf{cons}(\mathsf{fst}(sess), \mathsf{before})\ .[sess = sessCorrect].\\
& \mathsf{let}\ ksess = \mathsf{dec}(tsess, \mathsf{fst}(sess))\ .\mathsf{in}\ m\ .\mathsf{out}\ \mathsf{hash}(\mathsf{dec}(m, ksess))\ .Chk\\
Chk =& \mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ sess\ .\mathsf{nil}|AS'|C'|S'|DY)\\
DY =& \mathsf{repl.out}\ \mathsf{before}\ .\mathsf{out}\ \mathsf{after}\ .\mathsf{new}\ nonceDY\ .\mathsf{out}\ nonceDY\ .\mathsf{nil}|\\
& \mathsf{repl.in}\ x\ .\mathsf{in}\ y\ .\mathsf{out}\ \mathsf{cons}(x, y)\ .\mathsf{out}\ \mathsf{fst}(x)\ .\mathsf{out}\ \mathsf{snd}(x)\ .\\
& \mathsf{out}\ \mathsf{dec}(x, y)\ .\mathsf{out}\ \mathsf{enc}(x, y)\ .\mathsf{out}\ \mathsf{hash}(x)\ .\mathsf{out}\ \mathsf{val}(x)\ .\mathsf{out}\ \mathsf{next}(x)\ .\mathsf{nil}
\end{aligned}$$

where AS', C', S' are the same as AS, C, S except that before is replaced with after, msg1 is replaced with msg2, and Chk is replaced with nil. As in the Diffie-Hellman example, our specification, once exchanged a message msg1 or msg2, output its hash.

Using our tool, we generated $\mathcal{F}$ (77 states, 1424 transitions) and verified that $\mathsf{msg1} \notin [\texttt{@out}]_{\mathcal{F}}$ and $\mathsf{msg2} \notin [\texttt{@out-chk}]$, thus establishing the wanted properties. On a side note, we also have $\mathsf{msg1} \in [\texttt{@out-chk}]$, as it should be, since msg1 is actually disclosed and our analysis is sound.

6 A Bit of Compositionality

Real-world systems often run many different protocols in a concurrent fashion. However, one usually studies the security properties of each protocol independently. This may not be enough to ensure the integrity of a system, since two otherwise safe protocols may have unwanted interactions, especially if the protocols share secrets. One would rather be able to derive properties about $P_1|P_2$ from the studies of P_1 and P_2.

Our analysis offers some opportunities for composing security results. Assume P_1 and P_2 were analyzed beforehand, yielding the automata $\mathcal{F}_1$ and $\mathcal{F}_2$. We can build an $\mathcal{F}$ for $P_1|P_2$ by merging the transitions of $\mathcal{F}_1$ and $\mathcal{F}_2$ and adding

$$\begin{array}{ll}
\texttt{@in}_1 \to \texttt{@in} & \texttt{@in}_1 \to \texttt{@out}_2\\
\texttt{@in}_2 \to \texttt{@in} & \texttt{@in}_2 \to \texttt{@out}_1\\
\texttt{@out} \to \texttt{@out}_1 & \texttt{@out} \to \texttt{@out}_2
\end{array}$$

just as it happens for the analysis of the parallel. Such an $\mathcal{F}$ is sound, provided that $[\texttt{@out}_1]_{\mathcal{F}_1} \subseteq [\texttt{@in}_2]_{\mathcal{F}_2}$ and $[\texttt{@out}_2]_{\mathcal{F}_2} \subseteq [\texttt{@in}_1]_{\mathcal{F}_1}$. This last proof obligations might be checked by static analysis. If the obligations do not hold (or cannot be proved), the completion algorithm can be restarted from the above $\mathcal{F}$ to compute a sound approximation. This could be less expensive than rebuilding the approximation from scratch, since parts of the work have been already done when computing $\mathcal{F}_1$ and $\mathcal{F}_2$.

7 Conclusion

We presented a simple model for the specification of cryptographic protocols, based on process calculi and term rewriting. We stress that we allow *any* rewriting system for defining the cryptographic primitives. Further, the model deals with some basic temporal aspects, and therefore it is suitable to express certain security properties involving time, such as forward secrecy.

We defined a static analysis for the verification of protocols so that it is closed under rewritings. The analysis focuses on foreseeing the protocol behaviour before and after a selected point in time, represented by the firing of chk. Also, we explored some opportunities for composing results of our analysis.

We implemented the analysis, and used our tool to check some significant protocols. The tool confirmed that we can handle complex rewriting rules, such that those of exponentials, and protocols involving timestamps.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01), pages 104-115.*, 2001.
2. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999.
3. AVISPA project home page. http://www.avispa-project.org.
4. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, 2005.
5. C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for the π-calculus with application to security. *Journal of Information and Computation*, 168(1):68–92, 2001.
6. Y. Boichut. Tree automata for security protocols (TA4SP) tool. http:// lifc.univ-fcomte.fr/ boichut/TA4SP/TA4SP.html.
7. I. Cervesato, N. A. Durgin, J. C. Mitchell, P. D. Lincoln, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *13-th IEEE Computer Security Foundations Workshop*, pages 35–51, 2000.
8. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
9. D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, 1983.
10. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 2004.
11. T. Genet, Y. T. Tang-Talpin, and V. V. T. Tong. Verification of copy-protection cryptographic protocol using approximations of term rewriting systems. In *Proc. of Workshop on Issues in the Theory of Security*, 2003.
12. Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Proceeding of CADE*, pages 271–290, 2000.
13. Jean Goubault-Larrecq, Muriel Roger, and Kumar N. Verma. Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*, 64(2):219–251, August 2005.

14. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
15. J. K. Millen and V. Shmatikov. Symbolic protocol analysis with products and Diffie-Hellman exponentiation. In *Computer Security Foundations Workshop*, 2003.
16. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
17. David Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, 2003.
18. B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32:33–38, 1994.
19. F. Nielson, H. Riis Nielson, and H. Seidl. Cryptographic analysis in cubic time. *Electronic Notes in Theoretical Computer Science*, 62, 2002.
20. J. G. Steiner, B. C. Neuman, and J. I. Shiller. Kerberos: An authentication service for open network systems. In *Proc. of the Winter 1988 Usenix Conference*, pages 191–201, 1988.
21. Timbuk tree automata tool. http://www.irisa.fr/lande/genet/timbuk.
22. R. Zunino. Control flow analysis for the applied π–calculus. In *Proceedings of the MEFISTO Project 2003*, volume ENTCS 99, pages 87–110, 2004.
23. R. Zunino and P. Degano. Finite approximations of terms up to rewriting. http://www.di.unipi.it/ zunino/papers/completion.html.

Symbolic and Cryptographic Analysis of the Secure WS-ReliableMessaging Scenario*

Michael Backes[1], Sebastian Mödersheim[2], Birgit Pfitzmann[1], and Luca Viganò[2]

[1] IBM Zurich Research Lab, Switzerland
[2] Information Security Group, ETH Zurich, Switzerland

Abstract. Web services are an important series of industry standards for adding semantics to web-based and XML-based communication, in particular among enterprises. Like the entire series, the security standards and proposals are highly modular. Combinations of several standards are put together for testing as interoperability scenarios, and these scenarios are likely to evolve into industry best practices. In the terminology of security research, the interoperability scenarios correspond to security protocols. Hence, it is desirable to analyze them for security. In this paper, we analyze the security of the new Secure WS-ReliableMessaging Scenario, the first scenario to combine security elements with elements of another quality-of-service standard. We do this both symbolically and cryptographically. The results of both analyses are positive. The discussion of actual cryptographic primitives of web services security is a novelty of independent interest in this paper.

1 Introduction

Web services are a series of standards that add higher-layer semantics and quality of service to web-based communication. They use XML as the basic format for all exchanged content and SOAP as the basis for message exchanges [19]. In principle, web services are independent of the underlying transport protocol; in practice, as the name suggests, typical web protocols are commonly used. An important principle of web services is modularity (see [27]). This principle was in particular applied to the design of quality-of-service features like security and message ordering. Thus, these features are addressed by a set of standards and pre-standard proposals that can, at least syntactically, be combined in a highly flexible way. It is well-known, however, that combinations of security elements have to be treated with care as many combinations may not yield the properties that one might expect. The equivalent of the classic notion of security protocols in the web-services space is interoperability profiles or scenarios. While primarily defined for interoperability testing, they are not unlikely to evolve into industry best practices for common cases. At the same

* This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 428–445, 2006.
© Springer-Verlag Berlin Heidelberg 2006

time, they are at the level of concreteness where an analysis for well-known protocol security properties is possible.

In this paper, we present the first such analysis for an interoperability profile that combines features from the standards and proposals for security and another quality-of-service area, reliable messaging. It is the Secure WS-ReliableMessaging Scenario [24], which recently arose from the WS-ReliableMessaging and WS-SecureConversation Composability Interop Workshop held in April 2005.[1] It is based on the WS-Security standard [36] and the recent standard proposals WS-ReliableMessaging [28] and WS-SecureConversation [32] with a few additional references to WS-Trust [33] and WS-Addressing [18]. We present two types of analysis:

1. an automated analysis based on a number of symbolic protocol analysis techniques under the assumption of perfect cryptography, and
2. an analysis closer to real cryptography based on explicit cryptographic assumptions on the underlying cryptographic algorithms used.

Both analyses refer to the properties that are already informally stated in WS-ReliableMessaging [28], where they are pointed out as desirable security properties in the context of reliable sending of messages. WS-ReliableMessaging does not address how these properties can be achieved but refers to a suitable combination with the techniques offered by the security-specific web services standards. The Secure WS-ReliableMessaging Scenario provides such a combination, and our analysis exemplifies that the properties can indeed be achieved by the techniques offered by existing web services standards.

Our first, symbolic analysis has been carried out by employing the AVISPA Tool [2, 43], which is a push-button tool for the analysis of security-sensitive protocols and applications, under the assumption of perfect cryptography. The AVISPA Tool relies on a modular and expressive formal language for specifying protocols and their security properties, and integrates different back-ends that implement a variety of state-of-the-art automatic analysis techniques. For our analysis, we have employed OFMC [8] and CL-AtSe [42], which are the two more mature back-ends of the tool and which both perform protocol falsification and bounded verification by employing a number of symbolic techniques.

The Secure WS-ReliableMessaging Scenario has a structure that is far more complex than standard security protocols. Hence, an important part of modeling the protocol in a way feasible for automated analysis has been the search for a way to restrict the number of permissible interleavings of sending and receiving events without excluding attacks, i.e., every attack on the original protocol should be possible also on the simplified version. Below, we will first explain how we have built such a specification, and then illustrate the goals that we have checked in our analysis. Roughly speaking, we have shown that a client and a service mutually authenticate each other on certain messages that they exchange when executing the protocol, and that these messages remain secret.

[1] The title of [24] contains "scenarios" in the plural, but for our purposes the document defines one protocol and we thus use the singular.

These problems give rise to an infinite search space, so that automated tools need to make restrictions on some aspects of the problem in order to analyze it. We have considered different settings by imposing bounds on the number of possible parallel protocol sessions, on the number of message sequences that can be considered in each session, and on the number of payloads per message sequence. Neither OFMC nor CL-AtSe have reported any attacks for the settings we considered, and they have thus verified the Secure WS-ReliableMessaging Scenario with respect to the modeled security properties for these settings.

Our second analysis is manual (and thus more time-consuming, less flexible to protocol additions, and more prone to human error), but more realistic with respect to the cryptographic primitives. For instance, we show that we can treat the occurring key derivation via hash functions in the standard model of cryptography as pseudo-random functions if applied to certain pairs of arguments. For the other primitives, symmetric and asymmetric encryption as well as symmetric authentication and signatures, we can use standard definitions. We also discuss how close existing theorems on justifying symbolic analyses such as our first one come to replacing a from-scratch cryptographic analysis such as our second one. Note, however, that the Secure WS-ReliableMessaging Scenario, like all other current communication security standards, does not prescribe that provably secure primitives in the cryptographic sense are used, in particular for the symmetric primitives. Thus, we cannot claim that we proved exactly the standard implementations under what became known as standard cryptographic assumptions such as the hardness of factoring. Our cryptographic analysis is modular, and some results can immediately be reused for other profiles, e.g., the analysis of the initial key exchange based mainly on WS-Trust and that of the key derivation using elements of WS-SecureConversation.

Both our analyses have positive results, i.e., they demonstrate that at the abstraction level of each analysis, the protocol is error-free. Note that our two analyses are complementary (in particular, neither of them is derived from the other), but we consider it interesting future work to investigate how to link the two kinds of analysis for web services in the style of previous proofs of soundness of Dolev-Yao models, e.g., see [1, 5, 6, 7, 20, 40].

Outline of the Paper. We start by describing the Secure WS-ReliableMessaging Scenario and the corresponding security properties in Section 2. Sections 3 and 4 contain the symbolic and the cryptographic analysis of the scenario, respectively. After reviewing further related work in Section 5, we give concluding remarks and discuss possible future extensions of this work in Section 6.

Due to lack of space, discussions, examples, and proofs have been shortened or omitted; details can be found in the extended version of this paper [3].

2 The Secure WS-ReliableMessaging Scenario

The Secure WS-ReliableMessaging Scenario is a two-party protocol initiated by a client C and run together with a service S. It consists of three phases starting

Long-term keys:

pke_X, ske_X	Public and secret encryption key of $X \in \{C, S\}$.
pks_X, sks_X	Public and secret signature key of $X \in \{C, S\}$.
pks_{CA}	Public signature key of a certification authority CA.
$Cert_X$	Public key certificate of $X \in \{C, S\}$. We have $Cert_X = X, pke_X, pks_X, \mathsf{Sig}_{CA}(X, pke_X, pks_X)$, where $\mathsf{Sig}_{CA}(\cdot)$ denotes a signature computed by the certification authority CA, valid with respect to pks_{CA}.

Cryptographic primitives:

$\mathsf{Enc}_X(\cdot)$	A public-key encryption scheme, denoting encryptions computed with public key pke_X for $X \in \{C, S\}$.
$\mathsf{Sig}_X(\cdot)$	A digital signature scheme, denoting signatures computed with secret key sks_X for $X \in \{C, S\}$.
$\mathsf{SymEnc}_k(\cdot)$	A symmetric encryption scheme, denoting encryptions computed with secret key k.
$\mathsf{Mac}_k(\cdot)$	A message authentication code, denoting MACs computed with secret key k.
$\mathsf{Hash}(\cdot)$	A hash function, e.g., SHA-256.

Fig. 1. Keys and cryptographic algorithms used in the Secure WS-ReliableMessaging Scenario

Quantities occurring in the protocols:

$ID_1, \ldots, ID_9$	Message IDs of the individual protocol messages.
ID_{sk}	ID of the symmetric master key sk that is established in the initial key exchange phase.
ID_{Seq}	Sequence ID denoting the sequence of exchanged messages.
N, N^*	Nonces used to compute the master key sk.
N_1, N_2	Nonces used to compute the authentication and encryption session keys sk_1 and sk_2.
m	Payload that should be reliably sent from C to S.
n	Natural number denoting an acknowledged message.
k, k'	Symmetric keys used within a hybrid encryption in the initial key exchange phase.
sk	Symmetric master key shared between C and S after the initial key exchange phase. Derived from N and N^* as $sk = \mathsf{Hash}(N, N^*)$.
sk_1, sk_2	Symmetric session keys for authentication and encryption shared between C and S after the start of the message sending. Derived from sk, N_1, and N_2 as $sk_i = \mathsf{Hash}(N_i, sk)$.

Fig. 2. Quantities used in the Secure WS-ReliableMessaging Scenario

with a key-exchange phase, followed by the message-sending phase which uses this key, and finished by a termination phase which cancels the validity of the exchanged keys.

We will use a straight font to denote cryptographic algorithms (Enc, Sig, etc.), a straight font with capital letters to denote protocol-specific constants (RST, RSTR, etc.), and an *italic* font to denote keys, identities, etc.

The key-exchange phase is based on public-key cryptography and hence requires a mechanism to authenticate the respective public keys. The profile assumes a certification authority CA, which has a secret key sks_{CA}. Its public counterpart, pks_{CA}, is known to both C and S. The certification authority certifies the public keys of party $X \in \{C, S\}$ by signing the triple (X, pke_X, pks_X) with its key sks_{CA}, where pke_X and pks_X denote X's public encryption key and X's signature verification key, respectively. Note that pks_{CA} must have been conveyed in an authenticated manner to both C and S, and that pks_{CA} must not give certificates with the name X of an honest party to any other party.

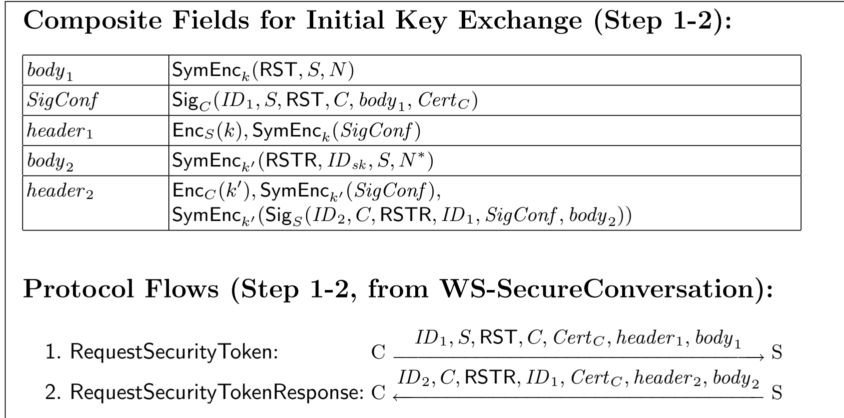
Figures 1 and 2 summarize the notation for the keys held by both parties, the cryptographic primitives we will be using, and the quantities involved in the protocol. For interoperability, the scenario uses specific cryptographic algorithms to implement the respective primitives — RSA-1.5 for public-key encryption, RSA-SHA1 for digital signatures, AES128-CBC for symmetric encryption, and HMAC-SHA1 for message authentication codes. In the cryptographic analysis that we carry out in Section 4, we do not fix specific algorithms but require that the used algorithms satisfy the respective security definitions under active attacks, e.g., indistinguishability under adaptive chosen-ciphertext attacks in the case of public-key encryption. Efficient schemes that satisfy these definitions exist under reasonable assumptions.

2.1 Description of the Protocol

Before the protocol begins, each party $X \in \{C, S\}$ has some starting information. Besides its own encryption and signature keys, the client starts with the signature verification key pks_{CA} of the certification authority CA, a certificate $Cert_C$ of its own public keys, and a certificate $Cert_S$ of the public keys of the service. The service starts only with the signature verification key pks_{CA} and with its own encryption and signature keys.

The protocol consists of nine steps, which we now briefly describe; an illustrative prose description of the individual steps based on Figures 3-5 is given in [3]. The first two steps constitute the key-exchange phase of the protocol between the client and the service and essentially rely on the functionalities offered by WS-SecureConversation; they are depicted in Figure 3. Similarly, the last two steps cancel the validity of this key as depicted in Figure 5. Steps three to seven are depicted in Figure 4 and constitute the message-sending phase, which consists of the creation of a message sequence, the secure sending of a message m, and the closing of the sequence; each of these steps essentially relies on the functionalities offered by WS-ReliableMessaging.

The protocol is not simply a ping-pong protocol: after the key-exchange phase has been completed, the client is allowed to start multiple sessions of the message-sending phase in parallel and there are non-deterministic choices on the order of messages.



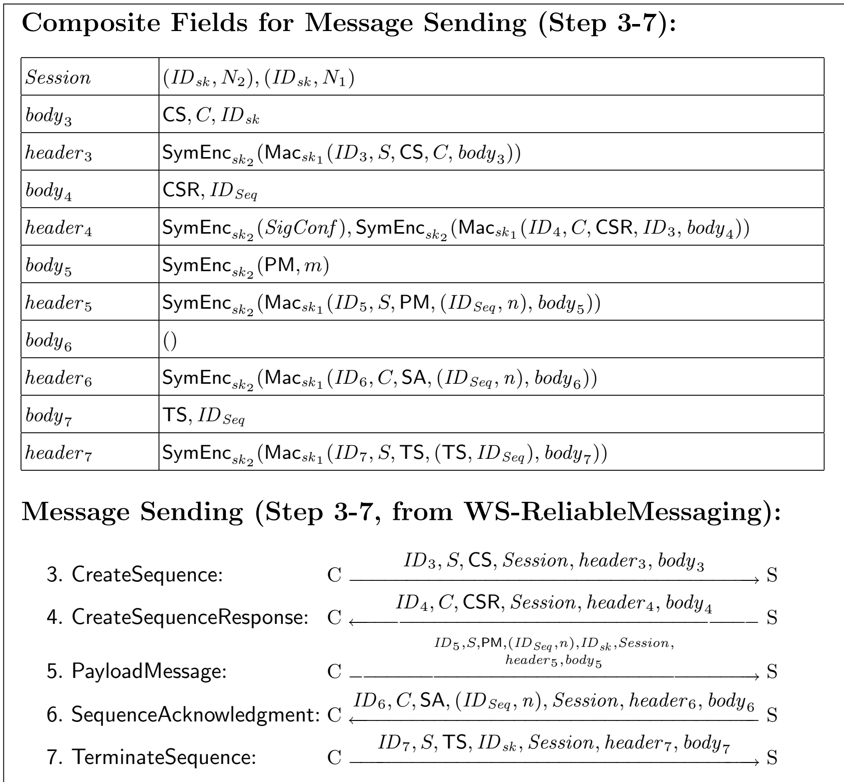
Composite Fields for Initial Key Exchange (Step 1-2):

$body_1$	$\mathsf{SymEnc}_k(\mathsf{RST}, S, N)$
$SigConf$	$\mathsf{Sig}_C(ID_1, S, \mathsf{RST}, C, body_1, Cert_C)$
$header_1$	$\mathsf{Enc}_S(k), \mathsf{SymEnc}_k(SigConf)$
$body_2$	$\mathsf{SymEnc}_{k'}(\mathsf{RSTR}, ID_{sk}, S, N^*)$
$header_2$	$\mathsf{Enc}_C(k'), \mathsf{SymEnc}_{k'}(SigConf)$, $\mathsf{SymEnc}_{k'}(\mathsf{Sig}_S(ID_2, C, \mathsf{RSTR}, ID_1, SigConf, body_2))$

Protocol Flows (Step 1-2, from WS-SecureConversation):

1. RequestSecurityToken: $C \xrightarrow{ID_1, S, \mathsf{RST}, C, Cert_C, header_1, body_1} S$
2. RequestSecurityTokenResponse: $C \xleftarrow{ID_2, C, \mathsf{RSTR}, ID_1, Cert_C, header_2, body_2} S$

Fig. 3. The Key-exchange Phase, implemented via WS-SecureConversation



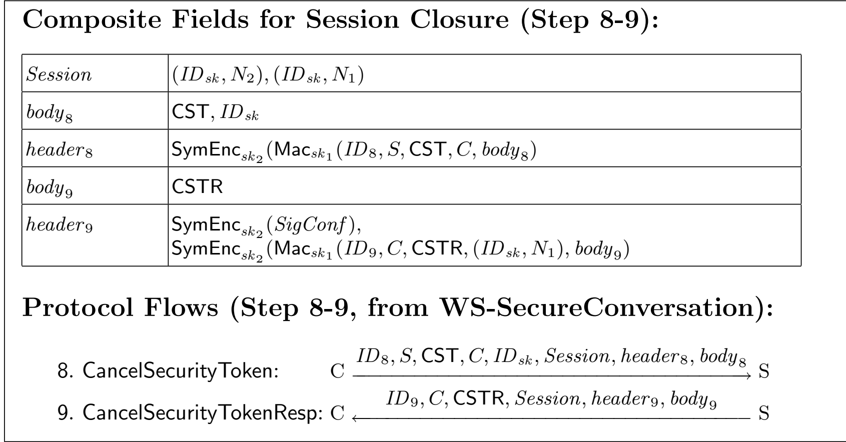
Composite Fields for Message Sending (Step 3-7):

$Session$	$(ID_{sk}, N_2), (ID_{sk}, N_1)$
$body_3$	CS, C, ID_{sk}
$header_3$	$\mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_3, S, \mathsf{CS}, C, body_3))$
$body_4$	CSR, ID_{Seq}
$header_4$	$\mathsf{SymEnc}_{sk_2}(SigConf), \mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_4, C, \mathsf{CSR}, ID_3, body_4))$
$body_5$	$\mathsf{SymEnc}_{sk_2}(\mathsf{PM}, m)$
$header_5$	$\mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_5, S, \mathsf{PM}, (ID_{Seq}, n), body_5))$
$body_6$	$()$
$header_6$	$\mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_6, C, \mathsf{SA}, (ID_{Seq}, n), body_6))$
$body_7$	TS, ID_{Seq}
$header_7$	$\mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_7, S, \mathsf{TS}, (\mathsf{TS}, ID_{Seq}), body_7))$

Message Sending (Step 3-7, from WS-ReliableMessaging):

3. CreateSequence: $C \xrightarrow{ID_3, S, \mathsf{CS}, Session, header_3, body_3} S$
4. CreateSequenceResponse: $C \xleftarrow{ID_4, C, \mathsf{CSR}, Session, header_4, body_4} S$
5. PayloadMessage: $C \xrightarrow{ID_5, S, \mathsf{PM}, (ID_{Seq}, n), ID_{sk}, Session, header_5, body_5} S$
6. SequenceAcknowledgment: $C \xleftarrow{ID_6, C, \mathsf{SA}, (ID_{Seq}, n), Session, header_6, body_6} S$
7. TerminateSequence: $C \xrightarrow{ID_7, S, \mathsf{TS}, ID_{sk}, Session, header_7, body_7} S$

Fig. 4. The Message-sending Phase, implemented via WS-ReliableMessaging



Composite Fields for Session Closure (Step 8-9):

$Session$	$(ID_{sk}, N_2), (ID_{sk}, N_1)$
$body_8$	CST, ID_{sk}
$header_8$	$\mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_8, S, \mathsf{CST}, C, body_8)$
$body_9$	CSTR
$header_9$	$\mathsf{SymEnc}_{sk_2}(SigConf)$, $\mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_9, C, \mathsf{CSTR}, (ID_{sk}, N_1), body_9)$

Protocol Flows (Step 8-9, from WS-SecureConversation):

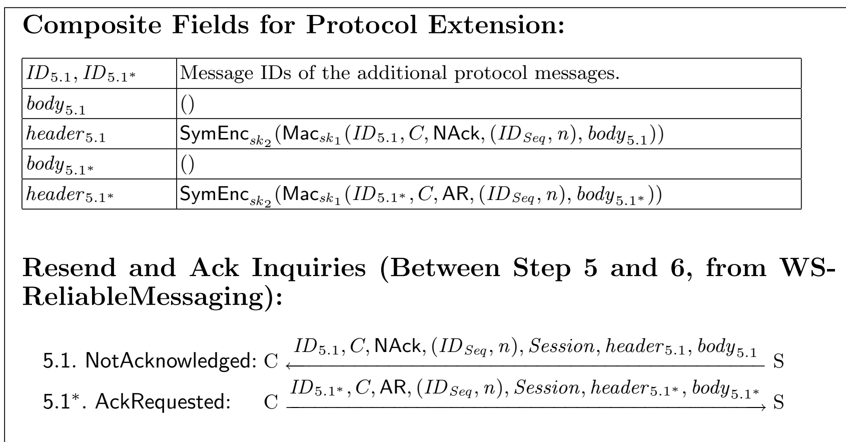
8. CancelSecurityToken: $\mathrm{C} \xrightarrow{ID_8, S, \mathsf{CST}, C, ID_{sk}, Session, header_8, body_8} \mathrm{S}$

9. CancelSecurityTokenResp: $\mathrm{C} \xleftarrow{ID_9, C, \mathsf{CSTR}, Session, header_9, body_9} \mathrm{S}$

Fig. 5. The Termination Phase, implemented via WS-SecureConversation

The necessary tests on the received messages follow the usual convention as described in [35], e.g., an honest receiver of a message checks that the decrypted plaintexts are of the correct format, that respective parts of the plaintext match corresponding parts sent unencrypted in the same message, and that the sender and receiver fields contain the expected values. We do not always mention this explicitly in the following.

Possible Protocol Extensions. We moreover sketch a possible extension of the interoperability scenario to reflect additional capabilities of the client and the service offered by the WS-ReliableMessaging standard. The standard



Composite Fields for Protocol Extension:

$ID_{5.1}, ID_{5.1^*}$	Message IDs of the additional protocol messages.
$body_{5.1}$	$()$
$header_{5.1}$	$\mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_{5.1}, C, \mathsf{NAck}, (ID_{Seq}, n), body_{5.1}))$
$body_{5.1^*}$	$()$
$header_{5.1^*}$	$\mathsf{SymEnc}_{sk_2}(\mathsf{Mac}_{sk_1}(ID_{5.1^*}, C, \mathsf{AR}, (ID_{Seq}, n), body_{5.1^*}))$

Resend and Ack Inquiries (Between Step 5 and 6, from WS-ReliableMessaging):

5.1. NotAcknowledged: $\mathrm{C} \xleftarrow{ID_{5.1}, C, \mathsf{NAck}, (ID_{Seq}, n), Session, header_{5.1}, body_{5.1}} \mathrm{S}$

5.1*. AckRequested: $\mathrm{C} \xrightarrow{ID_{5.1^*}, C, \mathsf{AR}, (ID_{Seq}, n), Session, header_{5.1^*}, body_{5.1^*}} \mathrm{S}$

Fig. 6. Extension of the Secure WS-ReliableMessaging Scenario with Resend Inquiries

additionally allows a client to request an unreceived acknowledgment of a previously sent message, and it allows a service to ask the client to re-send a message if it has not been received yet. This yields two additional steps which are depicted in Figure 6.

2.2 Security Properties

We consider a range of reasonable security requirements for the parties involved; some of the requirements are explicitly mandated by the standards, others are optional and hold only under stronger assumptions on the underlying cryptographic primitives. The following security properties are explicitly pointed out in WS-ReliableMessaging:

- *No Message Alteration*: Payloads contained in the 5. PayloadMessage in a session between an honest client and an honest service cannot be altered by an adversary.
- *No Message Disclosure*: Payloads contained in the 5. PayloadMessage in a session between an honest client and an honest service remain secret from the adversary.
- *Key Integrity and Confidentiality*: If an honest client and an honest service established a shared key *sk* after the first two steps of the protocol, both parties obtained the same key. Moreover, this key is secret from the adversary.
- *Authentication*: If an honest service accepts a payload m presumably from an honest client, then this honest client indeed sent this payload in the same session.

Accountability is also mentioned in WS-ReliableMessaging as one of the properties desirable in certain scenarios. As this scenario uses symmetric cryptography for the message authentication, accountability in the sense of non-repudiation is clearly not a goal of this scenario. The potential real-life accountability of this scenario is formally captured on the protocol level by the message integrity property and otherwise given by non-protocol factors. We refer to [3] for additional useful properties that are not explicitly required by the standard as well as for a refinement of the aforementioned properties tailored to the Secure WS-ReliableMessaging Scenario.

3 Symbolic Security Analysis

The AVISPA Tool. We have carried out a symbolic analysis of the Secure WS-ReliableMessaging Scenario by employing the *AVISPA Tool* [2, 43], which is a push-button tool for the automated validation, under the assumption of perfect cryptography and Dolev-Yao adversary [25], of industrial-scale Internet security-sensitive protocols and applications. A user interacts with the AVISPA Tool by specifying a *security problem* (a protocol paired with a security property that it is expected to achieve) in the *High-Level Protocol Specification Language*

HLPSL [21], which is an expressive, modular, role-based, formal language that allows for the specification of control-flow patterns, data structures, alternative adversary models, complex security properties, as well as different cryptographic operators and their algebraic properties. The AVISPA Tool automatically translates a user-defined security problem into an equivalent description of an infinite-state transition system that is then input to the back-ends of the AVISPA Tool. The back-ends search the transition system for states that represent attacks on the intended properties of the protocol.

The current version [2, 43] of the tool integrates four back-ends that implement a variety of state-of-the-art automatic analysis techniques, ranging from *protocol falsification* (by finding an attack on the input protocol) to *abstraction-based verification* methods for infinite numbers of sessions. The back-ends are: the *On-the-fly Model-Checker OFMC*, the *Constraint-Logic-based Attack Searcher CL-AtSe*, the *SAT-based Model-Checker SATMC*, and the *TA4SP verifier*, which analyzes protocols by implementing tree automata based on automatic approximations. All the back-ends of the tool analyze protocols by considering the standard Dolev-Yao model of an active adversary that controls the network but cannot break cryptography; in particular, the adversary can intercept messages and analyze them if it possesses the respective keys for decryption, and it can generate messages from his knowledge and send them under any party's name. Upon termination, the AVISPA Tool outputs that the protocol was verified with respect to the specified security problem, that an attack was found, or that the available resources were exhausted.

For our analysis of the Secure WS-ReliableMessaging Scenario, we have employed OFMC [8] and CL-AtSe [42], which are the two more mature back-ends of the tool, with better scope and performance. OFMC and CL-AtSe both perform protocol falsification and bounded verification by employing a number of symbolic techniques. Some of these techniques are back-end specific, while other ones are common to the two back-ends, such as the *lazy intruder technique* to symbolically represent all the possible messages that the Dolev-Yao adversary can generate. These techniques enable both OFMC and CL-AtSe to handle protocols with complex message terms and in particular to model the Secure WS-ReliableMessaging Scenario in its full complexity, without having to simplify the messages that are exchanged.[2]

The Model. The back-ends of the AVISPA Tool have successfully validated (or found a number of new attacks on) security protocols such as those in the Clark/Jacob library [22], as well as Kerberos, IKE, SET, and other protocols proposed by standardization organizations such as the IETF, ITU, W3C, Oasis, IEEE, 3GPP, and OMA. Similar analyses have been carried out by other (semi-)automated tools such as [9, 16, 17, 26, 41].

[2] The complexity of the Scenario prevents the usage of the current versions of SATMC and TA4SP. We hope to soon be able to report on the analysis with these back-ends as well; in particular, if analysis with TA4SP succeeded, then that would prove that the protocol is safe for secrecy goals for any number of sessions.

The Secure WS-ReliableMessaging Scenario has a structure that is far more complex than that of standard security protocols. Nonetheless, thanks to its expressiveness, HLPSL allows us to completely model the protocol, i.e., to provide a formal specification of the complex interactions between the two honest parties, which we can model as two separate client and service *programs* that communicate over an insecure network controlled by a Dolev-Yao adversary. However, such a model is too complex for automated analysis as even for a limited number of sessions, the set of permissible interleavings of sending and receiving events is enormous. For instance, the messages sent by the client may arrive in any order at the service. Additionally, both the client and the service can send "administrative" messages, i.e., acknowledge messages, request the retransmission of messages, or request the acknowledgment of messages. An important part of modeling the protocol in a way feasible for automated analysis has thus been the search for a way to restrict the number of interleavings without excluding attacks, i.e., such that any attack on the original protocol is possible also on the simplified version.

We have performed a step-by-step simplification of the client and service programs, whereby we have showed that these simplifications do not exclude any attacks.[3] As we lack space to give the HLPSL specification here due to its complexity and the amount of explanation that would be necessary, we only sketch the main ideas behind our HLPSL specification. In particular, we briefly illustrate the simplifications we have carried out for the client program; the ones for the service program are similar, and more details can be found in [3], together with a formal justification of the fact that these simplifications of the HLPSL specification do not exclude any attacks.

In order to simplify the client, note, firstly, that it is not a restriction if the client sends in one transition all the messages that it wishes to transmit via the 5. PayloadMessage step as soon as it has received the 4. CreateSequenceResponse message. Secondly, the client can neglect any requests of step 5.1. NotAcknowledged from the service to retransmit messages, since the Dolev-Yao adversary has seen all messages and can thus replay them to the service if this is necessary for an attack. Hence, we can consider a simplified client program that, having sent all its payload messages, simply waits for acknowledgment messages (6. SequenceAcknowledgment) or, after timing out, requests acknowledgment from the service (5.1*. AckRequested). Thirdly, since the Dolev-Yao adversary can intercept all responses from the service, it might deliberately make the client produce acknowledge request messages. Hence we can assume that the adversary can obtain acknowledge request messages of step 5.1*. AckRequested for every payload message. No attacks are therefore excluded if the client program sends with every 5. PayloadMessage also an 5.1*. AckRequested message.

[3] The simplified (restricted) version of the protocol that we obtain in this way is only useful for the formal analysis, not for the practical deployment of the protocol: for instance, since a Dolev-Yao adversary can replay old messages arbitrarily if this is necessary to mount an attack, we can restrict the model to client programs that never retransmit old messages.

These simplifications yield a client program that behaves as follows in every message sending phase: it sends all payload messages together with the corresponding requests for acknowledgment in one step, then waits until all messages are acknowledged, and finally sends a 7. TerminateSequence message.

Goals. Let us define the *security-relevant messages* of the Secure WS-ReliableMessaging Scenario to be the key-material (sk, sk_1, and sk_2) and all payloads transmitted with a 5. PayloadMessage. For our symbolic analysis, we have specified a number of *secrecy* and *authentication goals* (giving rise to different HLPSL security problems for the Scenario):

- secrecy of all security-relevant messages, and
- mutual authentication between client and service on all security-relevant messages.

We model these goals by labeling several transitions in the HLPSL specification with special events that express the meaning of the transition with respect to the goals of the protocol. First, whenever a client `c` that believes to talk with service `s` creates a security-relevant message `m`, then it generates a *secret* event `secret(m,{c,s})` expressing that `m` must remain secret between the parties in the specified set, in this case `c` and `s`. This allows us to define a *violation of secrecy* by a state of the transition system in which the adversary knows a message `m` for which a secrecy event has occurred with a set of parties to which the adversary does not belong. Second, we define violations of authentication by labeling the transitions with *witness* and *request* events. Whenever a party `a` that believes to talk with another party `b` first "handles" some security-relevant message `m` (i.e., either creates it or receives it for the first time), then it generates an event `witness(a,b,id,m)` where `id` is an identifier that uniquely determines the purpose of the message in the protocol. This witness event expresses that `a` uses message `m` for communication with `b` and for purpose `id`. The service `s` generates an event `request(s,c,id,m)` when it receives a payload `m` (supposedly) from the client `c` with index `id`. Similarly, if the client `c` receives the acknowledgement for the `id`-th payload (supposedly) from the service `s`, and if `c` has previously sent `m` as the `id`-th payload, then `c` generates the event `request(c,s,id,m)`. Similar request events are generated for the authentication on the key-material. (Intuitively, request events express that a party begins to rely on the agreement with another party on the specified value.)

A *violation of authentication* is then defined as any of the two following situations. First, *weak authentication* is violated whenever there is a `request(b,a,id,m)` but no matching witness event `witness(a,b,id,m)`, i.e., a party `b` believes a message `m` to come from `a`, but `a` has never sent `m`, at least not for this purpose. Second, *strong authentication* is violated whenever weak authentication is, or whenever a request event occurs more frequently than the corresponding witness event (i.e., by a kind of replay, the adversary made party `b` accept a message more often than it was actually said by `a`). Note that these goals are equivalent to Lowe's [39] notions of non-injective and injective agreement, respectively.

The security problems that we obtain by modeling these goals cover the main security properties stated for the Secure WS-ReliableMessaging Scenario in Section 2.2 as follows:

– secrecy of all security-relevant messages covers *no message disclosure* and *key confidentiality*,
– mutual authentication between client and service on all security-relevant messages covers *no message alteration*, *key integrity*, and *authentication.*

Bounds of the Analysis. The security problems associated with the Secure WS-ReliableMessaging Scenario give rise to an infinite search space, so that, in order to analyze this space, automated tools need to make some restrictions, i.e., to impose some bounds to consider relevant protocol execution and analysis settings. In the following, we will describe the restrictions that we imposed in our analysis with OFMC and CL-AtSe.

In general, there is no bound on the number of parties and sessions of the protocol that can be executed in parallel. While one can bound the number of parties, by the argumentations of [23] or by the *symbolic sessions technique* of OFMC [8], the problem of an unbounded number of sessions cannot be solved in general since it gives rise to undecidability. Moreover, there are two similar problems of unboundedness in the protocol: there is no bound on the number of payload messages to be exchanged or on the number of new message sequences that can be started, i.e., the protocol contains unbounded loops. All these problems give rise to an unbounded number of steps of honest parties, while both OFMC and CL-AtSe currently require analysis settings with bounded numbers of steps of honest parties.

In general, there is also no bound on the complexity of messages that the adversary can generate. However, as we remarked above, both OFMC and CL-AtSe implement the lazy intruder technique, which uses a symbolic representation to avoid explicitly enumerating the possible messages that the Dolev-Yao adversary can generate, and which allows for an analysis without restricting this parameter of the problem.

We have therefore analyzed the protocol with OFMC and CL-AtSe under the following execution/analysis settings: there are at most three parallel protocol sessions, the client can start at most two message-sending sequences per protocol session, and there are at most three payload messages per message sequence. Neither OFMC nor CL-AtSe have reported any attacks on the protocol for these analysis settings. In particular, for three parallel sessions, both OFMC and CL-AtSe verified the protocol within three hours (while the verification of smaller settings required between few seconds to a minute).

4 Cryptographic Security Analysis

In this section, we complement the symbolic analysis of the security properties of the WS-ReliableMessaging Scenario from Section 3 by a cryptographic analysis. Thus we now analyze the security of the scenario in a cryptographic setting

where the cryptographic primitives and the perfect cryptography assumption are replaced with actual cryptographic algorithms and the corresponding security notions that reason about probabilistic polynomial-time attackers. It is known that, even if the symbolic analysis is careful in distinguishing primitives like symmetric encryption and authentication, as both the analyzed scenario and the analysis in Section 3 do, and even if one assumes that an implementation is made with primitives secure according to the strictest usual cryptographic definitions, the results of such a symbolic analysis may not carry over to the real implementation. The most prominent example is that it cannot be avoided in general that the length of encrypted payload data, such as the values m in the PayloadMessage, leaks. Other problems that may occur in general scenarios are due to the probabilism of secure public-key encryption, key-stealing attacks, and manipulations of symmetric encryptions unless authenticated encryption [11, 10] is used in the implementation [5, 4]. Consequently, in a Dolev-Yao-style cryptographic library designed to be implemented based on arbitrary cryptographically secure primitives and to be usable in a secure way within arbitrary protocols with arbitrary security properties, both the abstraction and the realization must have certain idiosyncrasies. Hence, while it might be interesting to augment a tool like the AVISPA Tool by the idiosyncrasies of the Dolev-Yao style model of [5, 6, 4], and while implementing the primitives of WS-Security with the extended realizations from those papers (e.g., some additional tagging and randomization) might realize the goal of web service security to offer completely composable primitives also in a semantic sense, neither has been done yet. Other work on bridging the gap between symbolic and cryptographic security concentrated more on keeping very close to standard symbolic and real versions at the cost of generality. However, at present none of them covers the protocol class of Secure WS-ReliableMessaging, nor the security properties required. The seminal work [1] treats passive attacks only. Active attacks have been considered in this context in [40, 38, 20]. First, however, each of these papers treats only one cryptographic primitive, asymmetric encryption in [40, 20] and symmetric encryption in [38]. Secondly, [40] only treats integrity properties, while [38] only treats the secrecy of fixed, protocol-internal messages and [20] only treats the secrecy of nonces, i.e., random values chosen within the protocol and not usable for operations (such as encrypting) in that protocol. It may be interesting future work to extend such results on restricted usage of cryptographic libraries to the typical usage in WS-Security protocols. Our following considerations can be seen as a step in this direction.

Given these shortcomings of the current methods for deducing the security in the cryptographic setting from a symbolic proof, we do not try to do that, but base our proof directly on existing cryptographic work that explored the security of encryption, signatures, and MACs when combined in specific ways. In the following, we assume that the public-key encryption system Enc be secure against adaptive-chosen ciphertext attacks (short IND-CCA2-secure), that the symmetric encryption scheme be secure under adaptive chosen-plaintext attacks (short IND-CPA-secure), and that the signature scheme Sig and the message

authentication scheme Mac be secure against adaptive chosen-message attacks (short IND-CMA-secure). These are the commonly accepted security definitions of these primitives under active attacks so that we omit their rigorous definition. Primitives secure in this sense exist under reasonable assumptions.

Furthermore, we have to require that the hash function Hash used to compute the secret key *sk* based on two secret nonces does not degenerate the randomness induced by the nonces. This would be clear if we worked in the random oracle model; however, the specific setting of the scenario allows us to work in the standard model with a sufficient condition being that Hash, when applied to pairs, is a pseudo-random function in its first argument.

We obtain the following theorem (proven in [3]), in which we assume that the Secure WS-ReliableMessaging Scenario is run as a stand-alone protocol. This is not necessarily realistic for a web-services implementation; then our approach may have to take policies into account as in [14].

Theorem 1. *(Cryptographic Security of Secure WS-Reliable Messaging Scenario)* If Enc is IND-CCA2-secure, if Sig is IND-CMA-secure, if SymEnc is IND-CPA-secure, and if Hash, when applied to pairs, is a pseudo-random function in its first argument, then *key integrity and key confidentiality* are cryptographically fulfilled for the scenario, i.e., if the protocol is run with a probabilistic polynomial-time adversary, the keys are authentic with overwhelming probability, and the keys are indistinguishable from fresh random keys given the view of the adversary. If additionally Mac is IND-CMA-secure, then *message integrity* and *no message disclosure* are cryptographically fulfilled. □

5 Further Related Literature

Work is currently underway on scaling-up formal analysis methods and tools to web services security protocols, e.g., [12, 13, 14, 15, 29], although none of these works performs a cryptographic analysis of the protocols. In particular, the TulaFale tool [15] compiles descriptions of XML/SOAP-based security protocols and properties into the applied pi calculus and then employs the ProVerif tool [16]. We considered employing also TulaFale for the automatic symbolic analysis of Secure WS-ReliableMessaging, but its input language would first need to be extended to express all the constructs of the profile, and we thus leave this analysis and the comparison with our own symbolic analysis as future work. Recent work has also considered the automated analysis of XML-based web services: [37] presents a formal analysis of an encoding of the original XML messages into standard security protocol notation, showing that this encoding is without loss of attacks. Based on this encoding, the Casper/FDR tool can then check security properties for an unbounded number of sessions thanks to the employed data independence technique (which is similar to the abstraction techniques in TA4SP). The considered protocol, however, is simpler than the Secure WS-ReliableMessaging Scenario (e.g., no open-ended exchange of payload messages) and its analysis with Casper/FDR required simplifications of the

message terms. It is thus not clear if the method of [37] could also work on complex protocols such as the one considered in this paper.

Another type of analysis of a web services security protocol is that of an interoperability profile of WS-Federation in [31]. The analyzed profile [34] is a passive requestor profile, i.e., the user is represented only by a browser. The emphasis therefore lies on treating a browser in a protocol security proof. The analysis is by hand, and as only signatures and secure channels occur as cryptographic primitives, there is not much discussion of detailed properties of the cryptographic primitives in web services.

6 Conclusion and Outlook

We have given a symbolic and a cryptographic analysis of the security of the new Secure WS-ReliableMessaging Scenario, which constitutes the first web services scenario to combine security elements with elements of another quality-of-service standard. The results of both analyses are positive, i.e., they are proofs as far as the techniques faithfully represent the standards; these restrictions concern the cryptographic primitives and, in the symbolic case, the analysis settings. Our symbolic analysis is a further step in the use of formal proof tools for the validation of security protocols and web services under the perfect cryptography assumption. Our cryptographic analysis constitutes an important first step to reason about the security of web services in the more realistic setting where the perfect cryptography assumption is replaced by the complexity-theoretic definitions of cryptography. Some of the cryptographic results are of more general applicability in web services security than for the specific settings analyzed here.

As future work on the symbolic side, we have begun considering additional symbolic analysis settings, as well as employing abstraction techniques for carrying out unbounded verification. To this end, it would be particularly interesting not only to employ AVISPA's TA4SP, but also to investigate the relationships and possible complementarity of our analysis with an analysis carried out by TulaFale/ProVerif, especially since the model checkers that we used implement different techniques than those of ProVerif (which combines symbolic representations based on first-order logic and abstractions). Moreover, it would be of great help to be able to exploit the automatic compilation provided by TulaFale and we will thus investigate how to do so for the AVISPA Tool. We believe that the work of [30] will be helpful here, as it provides a preliminary translation procedure from protocol descriptions in HLPSL to descriptions in the applied pi calculus, which thus allows one to apply the ProVerif tool to some existing HLPSL protocol specifications.

On the cryptographic side, it would be interesting to see in which respect one can weaken the security requirements imposed on the cryptographic primitives without invalidating the security properties. Furthermore, we intend to apply our techniques to other profiles and scenarios and possibly even to a policy-based analysis similar to [14] on the symbolic side.

References

1. M. Abadi and P. Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Proc. 1st IFIP TCS*, LNCS 1872, pp. 3–22. Springer, 2000.
2. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proc. CAV'2005*, LNCS 3576, pp. 281–285. Springer, 2005.
3. M. Backes, S. Mödersheim, B. Pfitzmann, and L. Viganò. Symbolic and Cryptographic Analysis of the Secure WS-ReliableMessaging Scenario (Extended Version). Technical Report 502, Department of Computer Science, ETH Zurich, 2006. Available at `www.infsec.ethz.ch`.
4. M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Proc. 17th IEEE CSFW*, 2004.
5. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. 10th ACM CCS*, pp. 220–230, 2003. Full version in IACR Cryptology ePrint Archive 2003/015, Jan. 2003, `http://eprint.iacr.org/`.
6. M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In *Proc. 8th ESORICS*, LNCS 2808, pp. 271–290. Springer, 2003.
7. M. Backes, B. Pfitzmann, and M. Waidner. A general composition theorem for secure reactive systems. In *Proc. 1st TCC*, LNCS 2951, pp. 336–354. Springer, 2004.
8. D. Basin, S. Mödersheim, and L. Viganò. OFMC: A Symbolic Model-Checker for Security Protocols. *International Journal of Information Security*, 4(3):181–208, 2005.
9. G. Bella, F. Massacci, and L. C. Paulson. Verifying the SET Purchase Protocols. *Journal of Automated Reasoning*, to appear.
10. M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Proc. ASIACRYPT 2000*, LNCS 1976, pp. 531–545. Springer, 2000.
11. M. Bellare and P. Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient constructions. In *Proc. ASIACRYPT 2000*, LNCS 1976, pp. 317–330. Springer, 2000.
12. K. Bhargavan, R. Corin, C. Fournet, and A. Gordon. Secure sessions for web services. In *Proc. ACM Workshop on Secure Web Services (SWS)*, 2004.
13. K. Bhargavan, C. Fournet, and A. Gordon. A semantics for web service authentication. In *Proc. 31st POPL*, pp. 198–209. ACM Press, 2004.
14. K. Bhargavan, C. Fournet, and A. Gordon. Verifying policy-based security for web services. In *Proc. 11th ACM CCS*, pp. 268–277, 2004.
15. K. Bhargavan, C. Fournet, A. Gordon, and R. Pucella. TulaFale: A security tool for web servics. In *Proc. 2nd FMCO*, LNCS 3188, pp. 197–222. Springer, 2004.
16. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE CSFW*, pp. 82–96, 2001.
17. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.
18. D. Box, F. Curbera *et al.* Web Services Addressing (WS-Addressing), Aug. 2004.

19. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1, May 2000.
20. R. Canetti and J. Herzog. Universally composable symbolic analysis of cryptographic protocols (the case of encryption-based mutual authentication and key exchange). Cryptology ePrint Archive, Report 2004/334, 2004.
21. Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Proc. Workshop on Specification and Automated Processing of Security Requirements (SAPS'04)*, pp. 193–205. Austrian Computer Society, 2004.
22. J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997.
23. H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. In *Proc. 12th ESOP*, LNCS 2618, pp. 99–113. Springer, 2003.
24. D. Davis, C. Ferris, V. Gajjala, K. Gavrylyuk, M. Gudgin, C. Kaler, D. Langworthy, M. Moroney, A. Nadalin, J. Roots, T. Storey, T. Vishwanath, and D. Walter. Secure WS-ReliableMessaging scenarios, Apr. 2005. `ftp://www6.software.ibm.com/software/developer/library/ws-rmseconscenario.doc.`
25. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
26. B. Donovan, P. Norris, and G. Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proc. Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.
27. D. F. Ferguson, T. Storey, B. Lovering, and J. Shewchuk. Secure, reliable, transacted Web Services – architecture and composition, Oct. 2003. Available at `http://www-106.ibm.com/developerworks/webservices/library/ws-securtrans/.`
28. C. Ferris, D. Langworthy *et al.* Web Services Reliable Messaging Protocol (WS-ReliableMessaging), Feb. 2005.
29. A. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *Proc. 1st ACM Workshop on XML Security*, pp. 18–29, 2002.
30. A. Gotsman, F. Massacci, and M. Pistore. Towards an Independent Semantics and Verification Technology for the HLPSL Specification Language. *Electronic Notes in Theoretical Computer Science 135(1):59–77*, 2005.
31. T. Groß, B. Pfitzmann, and A.-R. Sadeghi. Proving a WS-Federation Passive Requestor Profile with a Browser Model. In *Proc. ACM Workshop on Secure Web Services (SWS)*, pp. 54–64. ACM Press, 2005.
32. M. Gudgin, A. Nadalin *et al.* Web Services Secure Conversation Language (WS-SecureConversation), Feb. 2005.
33. M. Gudgin, A. Nadalin *et al.* Web Services Trust Language (WS-Trust), Feb. 2005.
34. M. Hur, R. D. Johnson, A. Medvinsky, Y. Rouskov, J. Spellman, S. Weeden, and A. Nadalin. Passive Requestor Federation Interop Scenario, Version 0.4, Feb. 2004. `ftp://www6.software.ibm.com/software/developer/library/ws-fpscenario2.doc.`
35. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In *Proc. LPAR 2000*, LNCS 1955, pp. 131–160. Springer, 2000.
36. C. Kaler *et al.* Web Services Security (WS-Security), version 1.0, Apr. 2002.
37. E. Kleiner and A. Roscoe. On the relationship of traditional and Web Services Security protocols. *In Proceedings of the XXI Mathematical Foundations of Programming Semantics (MFPS'05). Electronic Notes in Theoretical Computer Science*, to appear.

38. P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proc. 25th IEEE Symposium on Security & Privacy*, pp. 71–85, 2004.
39. G. Lowe. A hierarchy of authentication specifications. In *Proc. 10th IEEE CSFW*, pp. 31–43, 1997.
40. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. 1st TCC*, LNCS 2951, pp. 133–151. Springer, 2004.
41. D. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9:47–74, 2001.
42. M. Turuani. *Sécurité des Protocoles Cryptographiques: Décidabilité et Complexité*. Phd, Université Henri Poincaré, Nancy, December 2003.
43. L. Viganò. Automated Security Protocol Analysis with the AVISPA Tool. *In Proceedings of the XXI Mathematical Foundations of Programming Semantics (MFPS'05). Electronic Notes in Theoretical Computer Science*, to appear.

Author Index

Abadi, Martín 398

Backes, Michael 428
Baldan, Paolo 126, 202
Baudet, Mathieu 398
Birkedal, L. 187
Blanqui, Frédéric 382
Bonsangue, Marcello M. 172
Bouajjani, Ahmed 94

Cao, Zining 63
Chen, Taolue 1
Corradini, Andrea 202
Şerbănuţă, Traian Florin 307

Debois, S. 187
Degano, Pierpaolo 413

Edalat, Abbas 231
Elsborg, E. 187

Fokkink, Wan 1
Francalanza, Adrian 16

Gorla, Daniele 47

Haar, Stefan 126
Heindel, Tobias 202
Hennessy, Matthew 16
Hildebrandt, T. 187

Jančar, Petr 277

Kirchner, Claude 382
König, Barbara 126, 202
Kurz, Alexander 172
Kuske, Dietrich 322

Laird, James 352
Laneve, Cosimo 32
Löding, Christof 292
Lohrey, Markus 322
Lüttgen, Gerald 261

Meyer, Antoine 94
Mödersheim, Sebastian 428

Nain, Sumit 1
Niss, H. 187

Ouaknine, Joël 217

Padovani, Luca 32
Palsberg, Jens 79
Paolini, Luca 367
Pattinson, Dirk 231
Pereira, Fernando Magno Quintão 79
Pfitzmann, Birgit 428
Phillips, Iain 246
Pimentel, Elaine 367
Pitcher, Corin 111
Popescu, Andrei 307

Rabinovich, Alexander 94
Riba, Colin 382
Riely, James 111
Rocca, Simona Ronchi Della 367
Roşu, Grigore 307
Rohde, Philipp 142

Sagiv, Mooly 94
Schröder, Lutz 157
Serre, Olivier 292, 337
Sobociński, Paweł 202
Srba, Jiří 277

Ulidowski, Irek 246

Viganò, Luca 428
Vogler, Walter 261

Warinschi, Bogdan 398
Worrell, James 217

Yorsh, Greta 94

Zunino, Roberto 413