Farhad Arbab
Marjan Sirjani (Eds.)

LNCS 8161

Fundamentals of Software Engineering

5th International Conference, FSEN 2013
Tehran, Iran, April 2013
Revised Selected Papers

Springer

Lecture Notes in Computer Science 8161

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

Editorial Board

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

For further volumes:
http://www.springer.com/series/558

Farhad Arbab · Marjan Sirjani (Eds.)

Fundamentals of Software Engineering

5th International Conference, FSEN 2013
Tehran, Iran, April 24-26, 2013
Revised Selected Papers

Springer

Volume Editors

Farhad Arbab
CWI Amsterdam, The Netherlands
E-mail: Farhad.Arbab@cwi.nl

Marjan Sirjani
Reykjavik University, Iceland
E-mail: marjan@ru.is

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40212-8 e-ISBN 978-3-642-40213-5
DOI 10.1007/978-3-642-40213-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013946950

CR Subject Classification (1998): F.3, D.2, F.1, D.4, F.4, C.2, J.7

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© IFIP International Federation for Information Processing 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

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

Preface

The present volume contains the proceedings of the 5th IPM International Conference on Fundamentals of Software Engineering (FSEN), held in Tehran, Iran, April 24–26, 2013. FSEN 2013 was organized by the School of Computer Science at the Institute for Research in Fundamental Sciences (IPM) in Iran, in cooperation with the ACM SIGSOFT and IFIP WG 2.2.

The topics of interest in FSEN span all aspects of formal methods, especially those related to advancing the application of formal methods in software industry and promoting their integration with practical engineering techniques. The Program Committee (PC) of FSEN 2013 consisted of 50 top researchers from 37 different academic institutes in 17 countries. We received 65 submissions from 33 countries, out of which the PC accepted 17 regular papers for the conference program. Each submission was reviewed by at least three independent referees, for its quality, originality, contribution, clarity of presentation, and its relevance to the conference topics.

Three distinguished keynote speakers delivered their lectures at FSEN 2013. Jose Meseguer gave a talk on "Symbolic Formal Methods: Combining the Power of Rewriting, Narrowing, SMT Solving and Model Checking." Holger Hermanns spoke on "Stochastic, Hybrid and Real-Time Systems: From Foundations to Applications with Modest." Wolfgang Reisig presented "Service-Oriented Computing: Forthcoming Challenges."

We thank the Institute for Research in Fundamental Sciences (IPM), Tehran, Iran, for their financial support and local organization of FSEN 2013. We thank the members of the PC for their time, effort, and contributions to making FSEN a quality conference. We thank Hossein Hojjat for his help in preparing this volume. Last but not least, our thanks go to our authors and conference participants, without whose submissions and participation FSEN would not have been possible.

June 2013

Farhad Arbab
Marjan Sirjani

Contents

Unbounded Allocation in Bounded Heaps 1
Jurriaan Rot, Frank de Boer, and Marcello Bonsangue

On the Complexity of Adding Convergence. 17
Alex Klinkhamer and Ali Ebnenasir

Deadlock Checking by Data Race Detection 34
Ka I Pun, Martin Steffen, and Volker Stolz

Delta Modeling and Model Checking of Product Families. 51
Hamideh Sabouri and Ramtin Khosravi

Lending Petri Nets and Contracts 66
Massimo Bartoletti, Tiziana Cimoli, and G. Michele Pinna

On Efficiency Preorders. 83
Manish Gaur and S. Arun-Kumar

Compiling Cooperative Task Management to Continuations 95
Keiko Nakata and Andri Saar

Extending UPPAAL for the Modeling and Verification
of Dynamic Real-Time Systems 111
*Abdeldjalil Boudjadar, Frits Vaandrager, Jean-Paul Bodeveix,
and Mamoun Filali*

Efficient Operational Semantics for ***EB***^{3} for Verification
of Temporal Properties 133
Dimitris Vekris and Catalin Dima

Interval Soundness of Resource-Constrained Workflow Nets:
Decidability and Repair 150
Elham Ramezani, Natalia Sidorova, and Christian Stahl

Statistical Model Checking of a Clock Synchronization
Protocol for Sensor Networks. 168
Luca Battisti, Damiano Macedonio, and Massimo Merro

A New Representation of Two-Dimensional Patterns
and Applications to Interactive Programming 183
*Iulia Teodora Banu-Demergian, Ciprian Ionut Paduraru,
and Gheorghe Stefanescu*

Push-Down Automata with Gap-Order Constraints 199
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Giorgio Delzanno, and Andreas Podelski

Model Checking MANETs with Arbitrary Mobility 217
Fatemeh Ghassemi, Saeide Ahmadi, Wan Fokkink, and Ali Movaghar

Validating SCTP Simultaneous Open Procedure 233
Somsak Vanit-Anunchai

Improving Time Bounded Reachability Computations in Interactive Markov Chains 250
Hassan Hatefi and Holger Hermanns

Checking Compatibility of Web Services Behaviorally 267
Kais Klai and Hanen Ochi

Author Index 283

Organization

General Chair

Hamid Sarbazi-azad	IPM, Iran; Sharif University of Technology, Iran

Steering Committee

Farhad Arbab	CWI, The Netherlands; Leiden University, The Netherlands
Christel Baier	University of Dresden, Germany
Frank de Boer	CWI, The Netherlands; Leiden University, The Netherlands
Ali Movaghar	IPM, Iran; Sharif University of Technology, Iran
Hamid Sarbazi-azad	IPM, Iran; Sharif University of Technology, Iran
Marjan Sirjani	Reykjavik University, Iceland
Jan Rutten	CWI, The Netherlands; Radboud University Nijmegen, The Netherlands

Program Chairs

Farhad Arbab	CWI, The Netherlands; Leiden University, The Netherlands
Marjan Sirjani	Reykjavik University, Iceland

Program Committee

Mohammad Abdollahi Azgomi	Iran University of Science and Technology, Iran
Gul Agha	University of Illinois at Urbana-Champaign, USA
Marco Aiello	University of Groningen, The Netherlands
Farhad Arbab	CWI and Leiden University, The Netherlands
Christel Baier	Technical University of Dresden, Germany
Jan Bergstra	University of Amsterdam, The Netherlands
Maria Paola Bonacina	Università degli Studi di Verona, Italy
Borzoo Bonakdarpour	University of Waterloo, Canada
Marcello Bonsangue	Leiden University, The Netherlands
Mario Bravetti	University of Bologna, Italy
Michael Butler	University of Southampton, UK
Frank De Boer	CWI and Leiden University, The Netherlands

Erik De Vink	Technische Universiteit Eindhoven, The Netherlands
Klaus Dräger	Oxford University, UK
Wan Fokkink	Vrije Universiteit Amsterdam, The Netherlands
Lars-Ake Fredlund	Universidad Politécnica de Madrid, Spain
Masahiro Fujita	University of Tokyo, Japan
Maurizio Gabbrielli	University of Bologna, Italy
Fatemeh Ghassemi	University of Tehran, Iran
Carlo Ghezzi	Politecnico di Milano, Italy
Jan Friso Groote	Eindhoven University of Technology, The Netherlands
Radu Grosu	Stony Brook University, USA
Hassan Haghighi	Shahid Beheshti University, Iran
Mohammad Izadi	Sharif University of Technology, Iran
Mohammad Mahdi Jaghoori	CWI, The Netherlands
Einar Broch Johnsen	University of Oslo, Norway
Joost-Pieter Katoen	RWTH Aachen, Germany
Narges Khakpour	KTH, Sweden
Ramtin Khosravi	University of Tehran, Iran
Joost Kok	Leiden University, The Netherlands
Kim Larsen	Aalborg University, Denmark
Zhiming Liu	United Nations University—International Institute for Software Technology, Macao
Sun Meng	Peking University, China
Hassan Mirian-Hosseinabadi	Sharif University of Technology, Iran
Ugo Montanari	Università di Pisa, Italy
Peter Mosses	Swansea University, UK
Mohammadreza Mousavi	Eindhoven University of Technology, The Netherlands
Ali Movaghar	Sharif University of Technology, Iran
Peter Olveczky	University of Oslo, Norway
Hiren D. Patel	University of Waterloo, Canada
Jose Proenca	Katholieke Universiteit Leuven, Belgium
Philipp Ruemmer	Uppsala University, Sweden
Jan Rutten	CWI and Radboud University Nijmegen, The Netherlands
Gwen Salaün	Grenoble INP—INRIA—LIG, France
Cesar Sanchez	IMDEA Software Institute, Spain
Davide Sangiorgi	University of Bologna, Italy
Wendelin Serwe	INRIA Rhône-Alpes/VASY, France
Marjan Sirjani	Reykjavik University, Iceland
Carolyn Talcott	SRI International, USA
Tayssir Touili	LIAFA, CNRS and University Paris Diderot, France

Local Organization

Hamidreza Shahrabi IPM, Iran

Proceedings Manager

Hossein Hojjat EPFL, Switzerland

Additional Reviewers

Attiogbe, Christian
Bacci, Giovanni
Balliu, Musard
Basold, Henning
Bentea, Lucian
Berg, Manuela
Bulanov, Pavel
Buscemi, Marzia
Chen, Zhenbang
Churchill, Martin
Corradini, Andrea
Cranen, Sjoerd
Dalla Preda, Mila
de Gouw, Stijn
Dubslaff, Clemens
Echenim, Mnacho
Emerencia, Ando
Faber, Johannes
Fox, Anthony
Fu, Hongfei
Gadducci, Fabio
Gerakios, Prodromos
Ghassemi, Fatemeh
Guan, Nan
Guanciale, Roberto
Hafez Qorani, Saleh
Harkjær Møller, Mikael
Helpa, Christopher
Helvensteijn, Michiel
Höftberger, Oliver
Isakovic, Haris
Jongmans, Sung-Shik T. Q.
Khamespanah, Ehsan
Khiri, Johan
Kokash, Natallia
Lampka, Kai
Lisser, Bert
Lluch Lafuente, Alberto
Macedo, Hugo
Madeira, Alexandre
Mauro, Jacopo
Mousavi, Mohammad Reza
Mukkamala, Raghava Rao
Nizamic, FarisParkinson, Matthew
Patrignani, Marco
Qamar, Nafees
Roohi, Nima
Salehi Fathabadi, Asieh
Sharma, Arpit
Snook, Colin
Soleimanifard, Siavash
Srba, Jiri
Subotic, Pavle
Tanhaei, Mohammd
Timmer, Mark
Torrini, Paolo
Wang, Shuling
Warriach, Ehsan
Wu, Stephen
Yautsiukhin, Artsiom
Ye, Lina

Invited Talks
(Abstracts)

Symbolic Formal Methods: Combining the Power of Rewriting, Narrowing, SMT Solving and Model Checking

Jose Meseguer

University of Illinois at Urbana-Champaign, Urbana, USA

Symbolic techniques that represent possibly infinite sets of states by symbolic constraints and support decision or semi-decision procedures based on such constraints have become essential to automate large parts of the verification effort and make verification much more scalable. They include: (i) SMT solving; (ii) rewriting- and unification-based techniques, including rewriting and narrowing modulo theories; and (iii) automata-based model checking techniques, which describe infinite sets of states and/or system traces symbolically by various kinds of automata. However, a key problem limiting the applicability of current symbolic techniques is lack of, or limited support for, extensibility. That is, although certain classes of systems can be formalized in ways that allow the application of specific symbolic analysis techniques, many other systems of interest fall outside the scope of such techniques. There is a real need to extend and combine the power of symbolic analysis techniques to cover a much wider class of systems. The talk will present some recent advances towards the goal of combined, extensible symbolic formal methods within the context of rewriting logic and Maude.

Stochastic, Hybrid and Real-Time Systems: From Foundations to Applications with Modest

Holger Hermanns

Saarland University–Computer Science,
Saarbrücken, Germany

Our reliance on complex safety-critical or economically vital systems such as networked automation systems or "smart" power grids increases at an everaccelerating pace. The necessity to study the reliability and performance of these systems is evident, but purely functional models and properties are insufficient in many cases. This has led to the development of integrative approaches that combine probabilities, real-time aspects and continuous dynamics with formal verification.

Today, formal quantitative modelling and analysis is supported by a wide range of tools and formalisms such as Prism with probabilistic guarded commands, Uppaal for graphical modelling and verification of timed automata, or hybrid system model checkers like Phaver. This variety of different languages and tools, however, is a major obstacle for new users seeking to apply formal methods in their field of work.

To overcome these problems, the Modest [4,6] modelling language and its underlying semantic model of stochastic hybrid automata (SHA) have been designed as an overarching formalism of which many well-known and extensively studied models such as Markov decision processes, probabilistic timed systems or hybrid automata are special cases. The construction and analysis of SHA models is supported by the Modest Toolset [1], which supports analysis with a range of different methods. At the current stage, the following analysis components are available: prohver [6] handles probabilistic safety properties for SHA; mcpta performs model checking of probabilistic timed automata using Prism; mctau [2] connects to Uppaal for model checking of timed automata, for which it is more efficient than mcpta; and modes [3] performs statistical model checking and simulation of stochastic timed automata with an emphasis on the sound handling of nondeterministic models.

The Modest Toolset has been used for a variety of applications with different levels of complexity and of expressiveness. These include *really cool* safety critical hard real-time wireless control applications for bicycles [5] as well

as high-speed trains [6], and innovative electric power grid control strategies [7]. The applications combine different abstraction and analysis techniques supported by the Modest Toolset.

Joint work with Arnd Hartmanns, Saarland University

References

1. The Modest Toolset website, `http://www.modestchecker.net`
2. Bogdoll, J., David, A., Hartmanns, A., and Hermanns, H.: mctau: Bridging the gap between Modest and UPPAAL. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 227–233. Springer, Heidelberg (2012)
3. Bogdoll, J., Hartmanns, A., and Hermanns, H.: Simulation and statistical model checking for Modestly nondeterministic models. In: Schmitt, J.B. (ed.) MMB & DFT 2012. LNCS, vol. 7201, pp. 249–252. Springer, Heidelberg (2012)
4. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., and Katoen, J.-P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Transactions on Software Engineering 32(10), 812–830 (2006)
5. Graf, H.B., Hermanns, H., Kulshrestha, J., Peter, J., Vahldiek, A., and Vasudevan, A.: A verified wireless safety critical hard real-time design. In: WOWMOM, pp. 1–9. IEEE (2011)
6. Hahn, E.M., Hartmanns, A., Hermanns, H., and Katoen, J.-P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods in System Design (2012)
7. Hartmanns, A., Hermanns, H., and Berrang, P.: A comparative analysis of decentralized power grid stabilization strategies. In: Winter Simulation Conference (2012)

Service Oriented Computing: Forthcoming Challenges

Wolfgang Reisig

Humboldt-Universität zu Berlin,
Berlin, Germany

Service-oriented Computing has established itself as a core paradigm of modern software architectures. Nevertheless, some obstacles prevent even more widespread use of service oriented architectures (SOAs). To overcome those obstacles, in particular the following questions have to be addressed:

1. SOAs are more and more implemented in the cloud. To what extent are the stakeholders affected by this change of technology?
2. It turned out useful to conceive not only software components, but also humans and technical systems as service providers and service requesters. How can a unified approach to SOA cope with this?
3. Basic notions such as correctness and equivalence are clear cut and undisputed for classical programs. Are there corresponding generally acceptable and manageable such notions for SOAs?
4. Quick assignment of needed data, software and hardware to services is inevitable for smoothly running SOAs. How can a small, flexible infrastructure guarantee this kind of elasticity?

Those questions cannot seriously be answered on an intuitive, informal level. It is inevitable to model services in a formal framework, with the decisive properties of the services be represented as properties of their formal models. The above questions are then addressed and faithfully solved in the framework of the formal models. To this end we suggest methods and principles of formally modeling and analyzing SOAs.

Unbounded Allocation in Bounded Heaps

Jurriaan Rot[1,2,⋆], Frank de Boer[1,2], and Marcello Bonsangue[1,2]

[1] LIACS—Leiden University, Leiden, Netherlands
[2] Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Netherlands
{jrot, marcello}@liacs.nl, frb@cwi.nl

Abstract. In this paper we introduce a new *symbolic* semantics for a class of recursive programs which feature dynamic creation and unbounded allocation of objects. We use a symbolic representation of the program state in terms of equations to model the semantics of a program as a pushdown system with a *finite* set of control states and a *finite* stack alphabet. Our main technical result is a rigorous proof of the equivalence between the concrete and the symbolic semantics.

Adding pointer fields gives rise to a Turing complete language. However, assuming the number of reachable objects in the visible heap is bounded in all the computations of a program with pointers, we show how to construct a program without pointers that simulates it. Consequently, in the context of bounded visible heaps, programs with pointers are no more expressive than programs without them.

1 Introduction

In this paper we investigate the interplay between dynamic creation of objects and recursion. To this end we introduce a core programming language which features dynamic object creation, global variables, static scope and recursive methods with local variables, but which does not include (abstract) pointers. In order to focus on the main issue of dynamic object creation in the context of recursion, we further restrict the data types to that of objects. Other *finite* data domains could have been added without problem, but would have increased the complexity of the model without strengthening our main result.

We first define a concrete operational semantics for our language based on a standard implementation of recursion using a stack. This semantics uses an explicit representation of objects which immediately gives rise to an infinite name space because an unbounded number of objects can be stored on the stack using local variables. Consequently, decidability results for pushdown systems (for which the stack alphabet is finite) and existing model checking techniques of pushdown systems against temporal formulas [2] are not applicable.

Our solution is to abstract from the concrete representation of objects by representing states, i.e., assignments of objects to the program variables, symbolically as conjunctions of equations over the program variables, identifying

⋆ The research of this author has been funded by the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920.

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 1–16, 2013.
DOI: 10.1007/978-3-642-40213-5_1,
© IFIP International Federation for Information Processing 2013

those variables which refer to the same object. In this symbolic setting we show how to describe the basic computation steps, e.g., object allocation, recursive calls and returns, in terms of a strongest postcondition semantics. The resulting symbolic semantics can be modeled formally as a pushdown system with a *finite* set of control states and a *finite* stack alphabet. Our main technical result is a rigorous proof of the equivalence between the concrete and the symbolic semantics.

Adding (abstract) pointers however allows to program dynamically linked data structures, like lists or trees, and allows the simulation of a 2-counter machine. As such, reachability is undecidable [9] for this extended language. We show in this paper that if a (recursive) program with pointers gives rise to computations in which the number of reachable objects in the heap is bounded a priori, then it can be simulated by a program without pointers. Therefore, in the context of bounded visible heaps, (recursive) programs with pointers are no more expressive than programs without them.

Related work. Our main contribution is a new symbolic semantics and a corresponding direct proof of decidability of a class of recursive programs which feature dynamic creation and unbounded allocation of objects, but do not include (abstract) pointers. The decidability of our core language itself follows from the general result of [3] for recursive programs with (abstract) pointers which generate only bounded visible heaps. On the other hand, the general result of [3] concerning pointers can be derived from our basic decidability result by our simulation of programs with pointers. In fact, our simulation shows that restricting to bounded (visible) heaps basically boils down to restricting to programs without pointers! Moreover, the general result of [3] is based on a complex mechanism for "merging" upon the return of a method the local part of the "old" heap (the heap before the call) and the current heap (see [11] for a more extensive discussion). This is because a reuse is required in order to model the semantics as a pushdown system with a finite stack alphabet (as explained above, by means of the local variables an unbounded number of objects can be stacked). In contrast, in the absence of pointers in our strongest postcondition semantics such name clashes are resolved symbolically by a simple substitution of fresh variables. The underlying equational logic then allows for a simple elimination of these implicitly existentially quantified variables.

More recently, [1] introduces an algorithm for reachability in pushdown automata with gap-ordered constraints, a model which allows to represent the behavior of our language. The symbolic semantics introduced in this paper is based on ordinary pushdown systems and therefore we can use standard algorithms for model checking [2]. Furthermore, it is similar in spirit to the one used in high level allocation Büchi automata [6] for model checking of a possibly unbounded number of objects with pointers but for a language with a *restricted form of recursion* (tail recursion) and *no block structure*. Full recursion, but with a *fixed-size number of objects* is instead considered in jMoped [7], using a pushdown structure to generate an infinite state system. Similar and even stronger restrictions limiting either size of heap and stack, or the number of objects are considered by current model checkers for object-oriented languages, such as Java

Path Finder [8], JCAT [5], Bandera [4] (possibly combined with the Symbolic Analysis Laboratory model checker [10]).

Plan of the paper. In Section 2 we introduce the syntax of our language and give an informal description of its semantics. Section 3 provides a concrete execution model using a transition system with infinite states and a symbolic model based on pushdown systems. We end the section by studying the relationship between these two models. In Section 4 we discuss the extension with pointers. Finally, the last section discusses some possible future research and tool development.

2 A Core Language for Allocation

We introduce a core programming language that focuses on dynamic allocation, global and local variables, and recursive procedures. To simplify the presentation it is restricted to a single data structure, that of objects. Objects can be dynamically allocated and referenced. A program consists of a finite set of procedures, each acting on some global and local state. Procedures can store object references in global or local variables, compare them, and call other procedures.

We assume a finite set of *program variables* V ranged over by x, y, v, w such that $V = G \cup L$, where G is a set of *global variables* $\{g_1, g_2, \ldots, g_n\}$ and L is a set of *local variables* $\{l_1, l_2, \ldots, l_m\}$, with G and L disjoint. We often write $\bar{l}$ for the sequence of all local variables. We assume a distinguished element $\mathsf{nil} \in G$, used as a constant to refer to the undefined object. For a finite set P of *procedure names* $\{p_0, \ldots, p_k\}$, a program is a set of *procedure declarations* of the form $p_i :: B_i$, where B_i, denoting the *body* of the procedure p_i, is a statement defined by the following grammar:

$$B ::= x := y \mid x := \mathsf{new} \mid \mathsf{call}\, p \mid B; B \mid [x = y]B \mid [x \neq y]B \mid B + B$$

The language is statically scoped. The *assignment* statement $x := y$ assigns the reference stored in y (if any) to x. The statement $x := \mathsf{new}$ *allocates* a new object that will be referenced by the program variable x. As for the ordinary assignment, the old value of x is lost. For this reason we will consider only programs in which the variable nil does not appear at the left-hand side of an assignment or allocation statement. *Sequential composition* $B_1; B_2$, *guarded statements* $[x = y]B$ and $[x \neq y]B$, and *nondeterministic choice* $B_1 + B_2$ have the standard interpretation. Execution of a *procedure call* $\mathsf{call}\, p$ consists of the execution of the associated body B with its local variables initialized to nil. Upon termination of the body B of a procedure the previous local state (from which the procedure has been called) is restored.

Example 1. As a basic example of storing an unbounded number of objects, consider the procedure

$$p :: l := \mathsf{new};\ ((\mathsf{call}\, p;\ g_1 := l) + g_2 := l)$$

A call to p allocates a number of objects not bounded a-priori and stores them in the several copies of the local variable l in the call-stack. The global variable

g_2 refers to the last allocated object (or nil), whereas g_1 refers to the first one (or nil). Note that if all variables are initially nil, then $g_1 \neq g_2$ eventually holds if and only if the program terminates.

The usual while, skip, if-then-else statements, and more general boolean expressions, can easily be encoded in this sequential setting. Procedures with call-by-value parameters and return values can also be modeled using assignments to global variables.

3 A Concrete and a Symbolic Semantics

In this section, we introduce a semantics of the programming language which is defined in terms of an explicit representation of objects by natural numbers. This representation allows a simple implementation of object allocation. A *program state* is a variable assignment $s\colon V \longrightarrow \mathbb{N}$, where 0 is used to represent the "undefined" object, and thus we assume $s(\mathsf{nil}) = 0$. To model allocation we distinguish a global "system" variable cnt which is used as a counter, and is not used by programs. We implicitly assume that $s(x) < s(\mathsf{cnt})$, for every state s and variable x different from cnt. Note that this implies that $s(\mathsf{cnt}) \neq 0$, as this is the value of nil.

A *configuration* of a program is a tuple $\langle s, S\rangle$ where s is the current program state and S is a stack of statements and stored return states. The current statement to be executed is on the top of the stack. An *execution step* of a program is a transition from a configuration C to a configuration C', denoted by $C \longrightarrow C'$. A computation is a (possibly infinite) sequence $C_1 \longrightarrow C_2 \longrightarrow \ldots$ of execution steps. The possible execution steps are given below. For modeling function updates we use multiple assignments of the form $f[x_1, \ldots, x_k := y_1, \ldots, y_k]$, where x_i and x_j are distinct elements of the domain of f for $i \neq j$, and all y_i's are in the codomain of f. It denotes the function mapping x_i to y_i if $i \in \{1, \ldots, k\}$, and otherwise x is mapped to the old value $f(x)$. The head of a stack is separated from the tail by means of the right-associative operator $\bullet$: for example, $B \bullet S$ is a stack consisting a statement B and tail S, whereas $s \bullet S$ is a stack consisting a state s as head and tail S.

When an *assignment* is the current statement to be executed, then the current program state is updated accordingly. The tail of the stack is not changed.

$$\langle s, x := y \bullet S\rangle \longrightarrow \langle s[x := s(y)], S\rangle$$

Dynamic allocation is similar to an assignment, but it uses the system variable cnt, which is now increased.

$$\langle s, x := \mathsf{new} \bullet S\rangle \longrightarrow \langle s[x, \mathsf{cnt} := s(\mathsf{cnt}), s(\mathsf{cnt}) + 1], S\rangle$$

The execution of the *sequential composition* of two statements updates the stack so that they are executed in the right order.

$$\langle s, B_1; B_2 \bullet S\rangle \longrightarrow \langle s, B_1 \bullet B_2 \bullet S\rangle$$

Guarded statements are executed only if the current program state satisfy their respective conditions, otherwise they block.

$$\frac{s(x) = s(y)}{\langle s, [x = y]B \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle} \qquad \frac{s(x) \neq s(y)}{\langle s, [x \neq y]B \bullet S\rangle \longrightarrow \langle s, B \bullet S\rangle}$$

A *non-deterministic choice* updates the stack so that only one of the two statements becomes the current one.

$$\frac{\langle s, B_i \bullet S\rangle \longrightarrow C}{\langle s, B_1 + B_2 \bullet S\rangle \longrightarrow C} \quad (i \in \{1, 2\})$$

In a *procedure call* the entire current state is pushed onto the stack so that the local variables can be restored when the procedure returns (we push the entire state only for notational convenience, to avoid irrelevant case distinctions). In the new current state all local variables are set to 0 and the current program on the stack becomes the body of the called procedure.

$$\langle s, \mathsf{call}\, p_i \bullet S\rangle \longrightarrow \langle s[\bar{l} := \bar{0}], B_i \bullet s \bullet S\rangle$$

Here B_i is the body of p_i, $\bar{l}$ denotes the sequence of local variables $l_1, \ldots, l_m$ and $\bar{0}$ is a sequence of length m of 0's, where m is the number of local variables.

When the top of the stack is a program state, a *procedure return* is executed. The local variables are restored using the state stored on the stack.

$$\langle s, s' \bullet S\rangle \longrightarrow \langle s[\bar{l} := s'(\bar{l})], S\rangle$$

Again, $\bar{l}$ denotes the sequence of local variables $l_1, \ldots, l_m$, while $s'(\bar{l})$ denotes the sequence of old values $s'(l_1), \ldots, s'(l_m)$.

3.1 A Symbolic Semantics

The concrete semantics introduced in the previous section clearly cannot be modeled as a pushdown system because it uses the infinite set of natural numbers to represent objects. However, at any moment of a computation, only finitely many objects are referenced by program variables. In this section we exploit this basic fact and represent a program state *symbolically* as a finite conjunction of equalities, identifying program variables referring to the same object. Subsequently, we define program steps based on a strongest postcondition calculus, which requires the introduction of fresh global variables for assignments. It suffices to assume a distinct logical variable z for each variable in V. We denote by Var the set of program variables V extended with their logical variables. Note that Var is finite. A *symbolic state* φ is a finite conjunction of equalities as given by the grammar

$$\varphi ::= x = y \;\mid\; \varphi \wedge \varphi$$

where x and y range over Var. A symbolic state φ gives rise to a relation $r(\varphi) \subseteq Var \times Var$, inductively defined by

$$r(x = y) = \{(x, y)\} \qquad r(\varphi_1 \wedge \varphi_2) = r(\varphi_1) \cup r(\varphi_2).$$

Next we let $r^*(\varphi) \subseteq Var \times Var$ denote the reflexive, symmetric and transitive closure of $r(\varphi)$. This way, any symbolic state φ gives rise to a partitioning of Var. We define $\varphi \models x = y$ if and only if $(x, y) \in r^*(\varphi)$, and extend it to disequalities by the closed world assumption, i.e., $\varphi \models x \neq y$ iff $\varphi \not\models x = y$.

We describe the effect of a basic statement S on a symbolic state φ in terms of its strongest postcondition $SP(S, \varphi)$, i.e., $SP(S, \varphi)$ will be the strongest state formula such that whenever we start from a program state satisfying φ (under the obvious satisfaction relation), after executing S the resulting state satisfies $SP(S, \varphi)$. We denote by $t[z/x]$ the syntactic substitution of x in t by a corresponding fresh (logical) variable z.

$$SP(x := \mathsf{new}, \varphi) = \varphi[z/x]$$
$$SP(x := y, \varphi) = \varphi[z/x] \wedge x = (y[z/x])$$

Note that z in the above two clauses represents the "old" value of x. In an assignment statement $x := y$ as well as in a dynamic allocation $x := \mathsf{new}$ we have for all variables v, w (syntactically) different from x that $\varphi \models v = w$ if and only if $SP(x := y, \varphi) \models v = w$. Further, for dynamic allocation, $SP(x := \mathsf{new}, \varphi) \models x = y$ if and only if x and y are the same variable (and thus by the closed world assumption $SP(x := \mathsf{new}, \varphi) \models x \neq y$, for every y different from x).

For procedure calls, we assume without loss of generality that for each global variable $g \in G$ there exists a unique *local* variable $g' \in L$ which does not occur in the given program. These so-called *freeze* variables are used to represent locally the global variables *before* a call. This information is needed to relate logically the state before the call and the return state. In the strongest postcondition of a procedure call we model logically the initialization of the local program variables (thus not the freeze variables) to nil, and the assignment to each freeze variable g' of the value of its corresponding global variable g:

$$SP(\mathsf{call}\, p, \varphi) = \varphi[\bar{z}/\bar{l}] \wedge \bigwedge_{g \in G} (g = g') \wedge \bigwedge_{l \in L'} l = \mathsf{nil},$$

where $\bar{z}$ is a sequence of logical variables corresponding to a sequence $\bar{l}$ of all local variables appearing in φ; $\bar{z}$ is used to represent the old values of $\bar{l}$. L' is the set of local variables that are not freeze variables.

To describe the strongest postcondition of the return of a procedure we introduce an auxiliary statement "$\mathsf{ret}\,\psi$" for each symbolic state ψ. The global variables in ψ thus represent the old values of the global variables *before* the call. On the other hand, in the current symbolic state φ the old equalities between the global variables before the call are represented by their corresponding freeze variables. We can identify these simply by replacing the freeze variables in φ and the global variables in ψ by the *same* logical variables. This explains the main idea underlying the following rule:

$$SP(\mathsf{ret}\,\psi, \varphi) = \varphi[\bar{z}/\bar{g}'][\bar{z}'/\bar{l}] \wedge \psi[\bar{z}/\bar{g}]$$

where $\bar{z}$ and $\bar{z}'$ are disjoint sequences of fresh logical variables, $\bar{g}'$ is the sequence of freeze variables and $\bar{l}$ is the sequence of all local variables. So in φ, first the

freeze variables are renamed to $\bar{z}$, and then the other local variables in φ, which are no longer valid, are renamed away into fresh logical variables $\bar{z}'$.

We use the above postcondition calculus to define a symbolic semantics for our programs. An (*abstract*) *configuration* of a program is a pair $\langle \varphi, \mathcal{S} \rangle$ where φ is a symbolic state *restricted* to the program variables V and $\mathcal{S}$ is a stack of statements and symbolic states also restricted to the program variables V. This restriction is justified because logical variables are implicitly existentially quantified and as such can be *eliminated* (in each step): for any symbolic state φ we can construct a formula $\varphi \downarrow_V$ which only contains variables in V such that $r^*(\varphi)$ restricted to $V \times V$ equals $r^*(\varphi \downarrow_V)$.

Now for dynamic allocation and assignment statements we lift the strongest postconditions defined above to transitions as follows:

$$\langle \varphi, B \bullet \mathcal{S} \rangle \longrightarrow \langle SP(B, \varphi) \downarrow_V, \mathcal{S} \rangle \tag{1}$$

where B is either $x := \mathsf{new}$ or $x := y$ for some program variables x and y.

The transition rules for sequential composition, non-deterministic choice and guarded statements are similar to the corresponding transition rules in the concrete semantics. As an illustration, we give below the two rules for guarded statements:

$$\frac{\varphi \models x = y}{\langle \varphi, [x = y]B \bullet \mathcal{S} \rangle \longrightarrow \langle \varphi, B \bullet \mathcal{S} \rangle} \qquad \frac{\varphi \models x \neq y}{\langle \varphi, [x \neq y]B \bullet \mathcal{S} \rangle \longrightarrow \langle \varphi, B \bullet \mathcal{S} \rangle} \tag{2}$$

On a procedure call p_i, we push the procedure body and current state onto the stack:

$$\langle \varphi, \mathsf{call}\, p_i \bullet \mathcal{S} \rangle \rightarrow \langle SP(\mathsf{call}\, p_i, \varphi) \downarrow_V, B_i \bullet \varphi \bullet \mathcal{S} \rangle. \tag{3}$$

The transition for procedure return is similar to that of an assignment:

$$\langle \varphi, \psi \bullet \mathcal{S} \rangle \rightarrow \langle SP(\mathsf{ret}\, \psi, \varphi) \downarrow_V, \mathcal{S} \rangle. \tag{4}$$

Example 2. As a simple example of a symbolic computation, consider the procedure declaration

$$p :: l := \mathsf{new}$$

where l is a local variable. Let g be some global variable. We will consider the execution of the statement $\mathsf{call}\, p$ starting from a symbolic state where g equals l. During the execution of p, l is assigned a new object; however, since l is a local variable, it is restored when the procedure returns, so then we should again have that g is equal to l. We restrict the above definition of the strongest postcondition of a procedure call to the local variable l and some global variable g; then $SP(\mathsf{call}\, p, l = g)$ is

$$z = g \wedge g = g' \wedge l = \mathsf{nil},$$

and thus by eliminating the logical variable z we derive the transition step

$$\langle l = g, \mathsf{call}\, p \rangle \longrightarrow \langle g = g' \wedge l = \mathsf{nil}, l := \mathsf{new} \bullet l = g \rangle.$$

Next we compute $SP(l := \mathsf{new}, g = g' \wedge l = \mathsf{nil})$:

$$g = g' \wedge z = \mathsf{nil},$$

and again eliminating the logical variable z we now derive the transition

$$\langle g = g' \wedge l = \mathsf{nil}, l := \mathsf{new} \bullet l = g \rangle \longrightarrow \langle g = g', l = g \rangle.$$

As above, restricting the above definition of $SP(\mathsf{ret}\, l = g, g = g')$ to the local variable l and the global variable g, we obtain

$$g = z \wedge l = z.$$

Finally, eliminating the logical variable z we arrive at the final transition

$$\langle g = g', l = g \rangle \longrightarrow \langle l = g, \epsilon \rangle,$$

where indeed l and g are again identified.

The above semantics gives rise to a finite *pushdown system.* A pushdown system is a triple $\mathcal{P} = (Q, \Gamma, \Delta)$ where Q is a finite set of *control locations*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is a finite set of *productions.* A transition $(q, \gamma, q', \bar{\gamma})$ is enabled if control is at location q and γ is at the top of the stack – then control can move to location q' by replacing γ by the possible empty word of stack symbols $\bar{\gamma}$.

In our case, for a given program $p_1 :: B_1, \ldots, p_n :: B_n$, the set of control locations is given by the set of state formulas restricted to V. In order to define the stack alphabet we introduce the finite set $\bigcup_{i=1}^{k} cl(B_i)$ of possible reachable statements where the closure of a statement B, denoted as $cl(B)$, is defined as follows:

$$cl(A) = \{A\} \qquad cl([x = y]B) = \{[x = y]B\} \cup cl(B)$$
$$cl(B_1; B_2) = \{B_1; B_2\} \cup cl(B_1) \cup cl(B_2) \qquad cl([x \neq y]B) = \{[x \neq y]B\} \cup cl(B)$$
$$cl(B_1 + B_2) = \{B_1 + B_2\} \cup cl(B_1) \cup cl(B_2)$$

where A is an assignment, an allocation or a procedure call. The stack alphabet Γ is then defined by the union of the abstract state space and the above set of possible reachable statements. Finally, it is straightforward to transform the rules of the above semantics into rules of a pushdown system, simply by removing the common stack tail from the left- and righthand sides. For a pushdown system both the halting problem and reachability are decidable. In fact, it is possible to model check pushdown systems against linear-time or branching-time temporal formulas. For linear-time temporal formulas the complexity is even of the same order as for finite state systems [2,7].

3.2 Correctness of the Symbolic Semantics

In this section we show that the concrete and the abstract semantics are equivalent. First we identify the relevant properties of the concrete semantics satisfied

by any reachable configuration. Basically, (1) on a procedure call, all local variables must be initialized to 0. Thus if an object is referenced by a variable in the current state s and in a stacked state s', then there must be a global variable referencing it in that stacked state s'. Moreover (2) the system variable cnt is greater than all objects currently referenced by variables stored somewhere in the stack. Formally this is the content of the next definition.

Definition 3. *A stack S is* proper *if either S is the empty stack, or $S = B \bullet S'$ for some statement B and proper stack S', or $S = s \bullet S'$ for some program state s and proper stack S' such that for any state s' occurring in S':*

$$\begin{array}{l}(1)\ s(V) \cap s'(V) \subseteq s'(G),\\ (2)\ \forall v \in V.s(\mathsf{cnt}) > s'(v).\end{array}$$

A configuration $\langle s, S\rangle$ is proper if $s \circ S$ is proper.

Properness is preserved by all computation steps:

Lemma 4. *If $\langle s, S\rangle$ is proper and $\langle s, S\rangle \rightarrow \langle s', S'\rangle$ then $\langle s', S'\rangle$ is proper.*

For example, the configuration $\langle s, p_0\rangle$ is proper, where p_0 is the main procedure name, and s is the state mapping cnt to 1 and all other variables (including nil) to 0. From the above Lemma, any configuration in a computation starting from this initial one is a proper configuration.

Next we give some basic properties of the concrete semantics of the procedure return. Informally, global variables are not affected by a procedure returns, and local variables get the values they had before the procedure call.

Lemma 5. *If $\langle s, s' \bullet S\rangle \longrightarrow \langle s_r, S\rangle$ then for every $x, y \in V$:*

1. $x, y \in G \Rightarrow (s_r(x) = s_r(y)$ *iff* $s(x) = s(y))$
2. $x, y \in L \Rightarrow (s_r(x) = s_r(y)$ *iff* $s'(x) = s'(y))$
3. $x \in G, y \in L \Rightarrow (s_r(x) = s_r(y)$ *iff* $s(x) = s'(y))$

We proceed with the following relevant properties of the symbolic semantics. First we make precise what is the relation between global variables in the caller's state ψ and freeze variables in the callee's state φ: freeze variables can be equal if and only if their corresponding global variables were equal in the first place.

Definition 6. *We define $\psi \triangleright \varphi$ iff for any two globals $g_1, g_2 \in G$,*

$$\varphi \models g_1' = g_2' \text{ iff } \psi \models g_1 = g_2$$

The definition of properness of abstract configurations is based on the above.

Definition 7. *A stack $\mathcal{S}$ (of symbolic states and statements) is* proper *if either $\mathcal{S}$ is the empty stack, or $\mathcal{S} = B \bullet \mathcal{S}'$ for some statement B and proper stack $\mathcal{S}'$, or $\mathcal{S} = \varphi \bullet \mathcal{S}'$ for some symbolic state φ and proper stack $\mathcal{S}'$ such that for the topmost state φ' occurring in $\mathcal{S}'$, if it exists: $\varphi' \triangleright \varphi$. An abstract configuration $\langle \varphi, \mathcal{S}\rangle$ is proper if $\varphi \bullet \mathcal{S}$ is proper.*

Properness of abstract configurations is preserved by all computation steps:

Lemma 8. *If $\langle \varphi, \mathcal{S} \rangle$ is proper and $\langle \varphi, \mathcal{S} \rangle \to \langle \varphi', \mathcal{S}' \rangle$ then $\langle \varphi', \mathcal{S}' \rangle$ is proper.*

We state the main properties of the procedure return in the abstract semantics.

Lemma 9. *If $\langle \varphi, \psi \bullet \mathcal{S} \rangle$ is proper, and $\langle \varphi, \psi \bullet \mathcal{S} \rangle \longrightarrow \langle \varphi_r, \mathcal{S} \rangle$ then for every $x, y \in V$:*

1. *$x, y \in G$ implies $\varphi_r \models x = y$ iff $\varphi \models x = y$,*
2. *$x, y \in L$ implies $\varphi_r \models x = y$ iff $\psi \models x = y$,*
3. *$x \in G$, $y \in L$ implies $\varphi_r \models x = y$ iff $\exists g \in G$ s.t. $\psi \models y = g$ and $\varphi \models x = g'$*

In order to prove the equivalence between the symbolic semantics and the concrete semantics, we extend the latter so that each procedure body starts with the initialization of the freeze variables to their corresponding global variables. Note that this does not affect the behaviour of a program, since freeze variables by assumption do not occur in it.

Let s be a concrete state, and φ a symbolic state. We define $s \sim \varphi$ if and only if for all variables $x, y \in V$: $s(x) = s(y)$ iff $\varphi \models x = y$. Next this relation is extended to stacks S of statements and (concrete) program states and stacks $\mathcal{S}$ of statements and symbolic states. We define $S \sim \mathcal{S}$ iff S and $\mathcal{S}$ are proper stacks, and one of the following cases hold:

1. S, $\mathcal{S}$ are both empty
2. $S = B \bullet S'$ and $\mathcal{S} = B \bullet \mathcal{S}'$ for some statement B, and $S' \sim \mathcal{S}'$
3. $S = s' \bullet S'$, $\mathcal{S} = \varphi \bullet \mathcal{S}'$, $s \sim \varphi$ and $S' \sim \mathcal{S}'$

Finally we lift the relation to *proper* configurations as follows: $\langle s, S \rangle \sim \langle \varphi, \mathcal{S} \rangle$ iff $s \bullet S \sim \varphi \bullet \mathcal{S}$.

In our setting a *bisimulation* is a relation R, between concrete- and abstract configurations, such that for every $(C, D) \in R$: if $C \longrightarrow C'$ then there is a configuration D' such that $D \longrightarrow D'$ and $(C', D') \in R$, and vice versa; where $C \longrightarrow C'$ is a transition between concrete configurations, given by the concrete semantics (augmented with the initialization of freeze variables), and $D \longrightarrow D'$ is a transition between abstract configurations, given by the symbolic semantics. Our main result states the equivalence between the operational and the symbolic semantics, based on this notion.

Theorem 10. *The above relation $\sim$ between proper configurations is a (strong) bisimulation.*

Proof. Suppose $\langle s, S \rangle \sim \langle \varphi, \mathcal{S} \rangle$. The proof proceeds by cases on top of the stacks, i.e., the program constructs; we only treat the case of procedure return. In this case $S = s' \bullet S'$ and $\mathcal{S} = \psi \bullet \mathcal{S}'$ for some s' and ψ, and there are s_r and φ_r such that

$$\langle s, s' \bullet S' \rangle \longrightarrow \langle s_r, S' \rangle \text{ and } \langle \varphi, \psi \bullet \mathcal{S}' \rangle \longrightarrow \langle \varphi_r, \mathcal{S}' \rangle$$

Note that we have $s' \sim \psi$, $s \sim \varphi$ and $S' \sim \mathcal{S}'$ by assumption. Further note that we can apply Lemma 9 since by assumption $\langle \varphi, \mathcal{S} \rangle$ is proper. We must show that $s_r \sim \varphi_r$ holds. Let $x, y \in V$. We distinguish three cases:

1. $x, y \in G$ (both global variables):

$$\begin{array}{llll} s_r(x) = s_r(y) & \text{iff} & s(x) = s(y) & \text{Lemma 5.1} \\ & \text{iff} & \varphi \models x = y & \text{assumption} \\ & \text{iff} & \varphi_r \models x = y & \text{Lemma 9.1} \end{array}$$

2. $x, y \in L$ (both local variables):

$$\begin{array}{llll} s_r(x) = s_r(y) & \text{iff} & s'(x) = s'(y) & \text{Lemma 5.2} \\ & \text{iff} & \psi \models x = y & \text{assumption} \\ & \text{iff} & \varphi_r \models x = y & \text{Lemma 9.2} \end{array}$$

3. $x \in G$ and $y \in L$. Recall that we have augmented the concrete semantics with the initialization of the freeze variables. It is easy to see that as a consequence $s(g') = s'(g)$ holds, which we will use below.

$$\begin{array}{lll} & s_r(x) = s_r(y) & \\ \text{iff} & s(x) = s'(y) & \text{Lemma 5.3} \\ \text{iff} & \exists g \in G. s(x) = s'(g) \text{ and } s'(g) = s'(y) & \text{properness} \\ \text{iff} & \exists g \in G. s(x) = s(g') \text{ and } s'(g) = s'(y) & \text{above argument} \\ \text{iff} & \exists g \in G. \varphi \models x = g' \text{ and } \psi \models g = y & \text{assumption} \\ \text{iff} & \varphi \models x = y & \text{Lemma 9.3} \end{array}$$

□

4 Adding Pointers

In this section we extend our programming language with fields and corresponding updates for modeling linked object structures. In particular we investigate k-*bounded heaps*, in which the number of reachable objects is at most k. This notion is extended to k-bounded programs, in which during execution, the heap is always k-bounded. We show that every k-bounded program P including fields can be rewritten into a program P' without fields and equivalent to P.

The language of Section 2 is extended with statements $x := y.f$ and $x.f := y$, where f ranges over a finite set of (pointer) fields F. Note that the only type of field is that of a pointer to another object. Informally, the statement $x := y.f$ is a basic assignment, updating x to point to (the location referred to by) $y.f$. Field update $x.f := y$ changes to y the field of the object to which x refers via the field f. These two basic operations are sufficient; more general expressions and updates can be encoded. For example, a statement $x := y.f_{i_1}.f_{i_2}.f_{i_3} \dots f_{i_k}$ is encoded as $x := y.f_{i_1}; x := x.f_{i_2}; x := x.f_{i_3}; \dots ; x := x.f_{i_k}$.

To give a semantics to this language we introduce a *heap* H as a pair $\langle s, h\rangle$ of a variable assignment $s : V \to \mathbb{N}$ such that $s(\mathsf{nil}) = 0$, and a field assignment $h : F \to (\mathbb{N} \to \mathbb{N})$ such that for all f, $h(f)(0) = 0$. We write $H(x)$ for $s(x)$, and $H(f)$ for $h(f)$.

For a set of variables X we denote with $\mathcal{R}_H(X) \subseteq \mathbb{N}$ the set of objects reachable from these variables in H, defined as the least fixpoint of the equation

$$\mathcal{R}_H(X) = \{H(x) \mid x \in X\} \cup \{H(f)(n) \mid f \in F, n \in \mathcal{R}_H(X)\}$$

We abbreviate $\mathcal{R}_H(V)$, where as before V denotes the set of program variables, by $\mathcal{R}_H$. We denote an assignment to a variable by $H[x := n]$, a global field update by $H[f := \rho]$ with $\rho : \mathbb{N} \to \mathbb{N}$ s.t. $\rho(0) = 0$, and local field update by $H[f := \rho[n := m]]$. We use the standard notation and definition of simultaneous assignments and updates.

A configuration is a tuple $\langle H, \Gamma \rangle$ where H is a heap and Γ is a stack of statements and heaps. We only give the transitions for dynamic allocation, and for assignments of the form $x.f := y$ and $x := y.f$. All other transitions are similar to the semantics of the language without fields, and are not repeated here. Again we assume a system variable cnt for the implementation of dynamic allocation. On a *dynamic allocation* all fields of the new object are set to point to 0.

$$\langle H, x := \mathsf{new} \bullet \Gamma \rangle \longrightarrow \langle H[x, \mathsf{cnt} := H(\mathsf{cnt}), H(\mathsf{cnt}) + 1][\bar{f} := \bar{\rho}], \Gamma \rangle$$

where $\bar{\rho}$ is a sequence such that $\rho_i = H(f_i)[H(x) := 0]$. A *field update* $x.f := y$ is as follows:

$$\frac{H(x) \neq H(\mathsf{nil})}{\langle H, x.f := y \bullet \Gamma \rangle \longrightarrow \langle H[f := H(f)[H(x) := H(y)], \Gamma \rangle}$$

Finally an assignment of the form $x := y.f$ is as follows:

$$\frac{H(y) \neq H(\mathsf{nil})}{\langle H, x := y.f \bullet \Gamma \rangle \longrightarrow \langle H[x := H(f)(H(y))], \Gamma \rangle}$$

It is not hard to see that reachability is undecidable for this language. For instance, we can simulate a 2-counter machine [9] by using three variables c_1, c_2, t and a single field f. An increment of c_i is then implemented as a statement $t := \mathsf{new}; t.f := c_i; c_i := t$, a decrement is a simple $c_i := c_i.f$ and we can test for zero by using a guard $[c_i = \mathsf{nil}]$. It is thus not possible to devise, in the same fashion as in Section 3, a precise abstraction of the semantics of our extended language for which reachability is decidable.

In [3] an abstraction of heaps in terms of isomorphic graphs is given, which is applied in a semantics with the property that reachability is decidable for programs in which every heap is k-bounded for some a priori fixed k.

Definition 11. *A heap H is k-bounded if $|\mathcal{R}_H| \leq k$. A computation $\langle H_1, \Gamma_1 \rangle \to \langle H_2, \Gamma_2 \rangle \to \ldots$ is k-bounded if $|\mathcal{R}_{H_i}|$ is k-bounded for all i. A program P with main procedure p_0 is k-bounded if every computation $\langle H_0, p_0 \rangle \to \ldots$ is k-bounded, for some initial heap H_0.*

We show that k-bounded programs with fields can be simulated by programs without fields in our basic language. We do so by means of a transformation from k-bounded programs to equivalent programs not containing any fields.

We first show how to represent k-bounded heaps using only plain variable assignments of the form $s : V \to \mathbb{N}$. To this end let k be a given bound. The correspondence between a k-bounded heap H and a state s is based on an explicit

enumeration of the objects in the visible heap, represented by k global variables $\bar{1}, \ldots, \bar{k}$ such that $s(\bar{i}) \neq s(\bar{j})$, for $i \neq j$. Notice that the undefined object 0 is already represented by the variable nil, and as such we have, in fact, a series $\mathsf{nil}, \bar{1}, \ldots, \bar{k}$ of $k+1$ variables to represent k objects, which will turn out to be of technical convenience. Further, we introduce for each $i = 1, \ldots, k$ and field $f \in F$ a global variable $\bar{i}_f$ which represents $H(f)(s(\bar{i}))$. Without loss of generality we assume that these variables do not appear in the given program. In the sequel we denote by I the set of global variables $\bar{1}, \ldots, \bar{k}$ and by $V(P)$ the (global and local) variables which do occur in P. We next define when a k-bounded heap H is represented by a variable assignment s, denoted by $H \equiv s$.

Definition 12. *Given a k-bounded heap and a variable assignment s we define $H \equiv s$ by*

1. $s(\bar{i}) \neq s(\bar{j})$ *and* $s(\bar{i}) \neq 0$, *for* $i \neq j$, $i = 1, \ldots, k$ *and* $j = 1, \ldots, k$,
2. *for every* $n \in \mathcal{R}_H(V(P))$ *s.t.* $n \neq 0$ *there is* $i = 1, \ldots, k$ *such that* $s(\bar{i}) = n$,
3. $H(f)(s(\bar{i})) = s(\bar{i}_f)$, *for every* $i = 1, \ldots, k$ *and* $f \in F$,
4. $H(\mathsf{cnt}) = s(\mathsf{cnt})$.

The actual program transformation now is defined in terms of a function t which takes a k-bounded program P with fields and translates it into an equivalent program $t(P)$ without fields.

For each of the procedures except the initial one, we define $t(p_i :: B_i) = p_i :: t(B_i)$, where $t(B)$ will be defined by structural induction. We append in front of the initial procedure $p_0 :: B_0$ a series of allocations and assignments to initialize the representation of the heap:

$$\begin{aligned} p_0 :: \; & \Pi_{\bar{i} \in I}(\bar{i} := \mathsf{new}; \Pi_{f \in F} \bar{i}_f := \mathsf{nil};) \\ & \mathsf{true} := \mathsf{new}; \mathsf{false} := \mathsf{new}; \\ & t(B_0) \end{aligned}$$

where the *for all*-construct $\Pi_{l \in L} B$ is a shorthand for a sequential composition of the statements $B_{l'}$, for $l' \in L$, where $B_{l'}$ is obtained from B by substituting l' for l in B. For a better readability we omit the parenthesis in a for-all construct and assume its scope is clear from the context. The fresh global variables true and false, which are assumed not appear in the given program, will be used to encode boolean values.

We proceed to discuss for each of the statements in the language its translation. A simple assignment $x := y$ remains unchanged and the translation of a sequential composition or choice between two statements consists of the sequential composition or choice between the translated statements. Since the conditional statements refer only to plain variables, we simply have

$$t([x = y]B) = [x = y]t(B).$$

For an assignment of the form $x := y.f$, in order to find the variable representation of $y.f$, we must find the variable $\bar{i}$ which represents the object denoted by y.

The variable $\bar{i}_f$ then represents $y.f$. This search and corresponding assignment is described simply by the following non-deterministic choice:

$$t(x := y.f) = \Sigma_{\bar{i} \in I}[\bar{i} = y]x := \bar{i}_f.$$

Note that this "n-ary" non-deterministic choice generalizes binary choice in the usual manner. A field update $x.f := y$ is treated in a similar way:

$$t(x.f := y) = \Sigma_{\bar{i} \in I}[\bar{i} = x]\bar{i}_f := y.$$

Notice that both for field updates $x.f := y$ and $x := y.f$, the condition $H(x) \neq H(\mathsf{nil})$ of the semantic rules are enforced by only considering variables from I, which represent non-null objects.

To simulate a dynamic allocation $x := \mathsf{new}$, we non-deterministically select a variable $\bar{i} \in I$ which denotes an object that is *not* reachable from any variables in $V(P)$. This variable will be (re)used to represent the newly created object assigned to x. Reachability in a k-bounded heap can be implemented by the following statement using for each $\bar{i} \in I$ a fresh variable $\bar{i}_b$ to indicate that $\bar{i}$ is reachable from the variables in X:

$$R_X = \Pi_{\bar{i} \in I}\bar{i}_b := \mathsf{false};\ \Pi_{x \in X}\Sigma_{i=1}^{k}[\bar{i} = x]\bar{i}_b := \mathsf{true};\ B^k$$

where B denotes the statement

$$\Pi_{\bar{i} \in I}([\bar{i}_b = \mathsf{false}]\mathsf{skip} + [\bar{i}_b = \mathsf{true}]\Pi_{f \in F}\Sigma_{\bar{j} \in I}[\bar{i}_f = \bar{j}]\bar{j}_b := \mathsf{true})$$

and B^k denotes the sequential composition of k copies of B. Note that because the visible heap is k-bounded we need to iterate the statement B only k times. Further notice that since the undefined object is always reachable by the variable nil, and the heap is k-bounded, there can only be $k-1$ other reachable objects. So there will always be a representative $\bar{i} \in I$ of a non-null object which is not reachable, i.e., $\bar{i}_b = \mathsf{false}$ after executing the above reachability algorithm. This motivates the following translation of object creation:

$$t(x := \mathsf{new}) = R_{V(P)}; \Sigma_{\bar{i} \in I}([\bar{i}_b = \mathsf{false}]\bar{i} := \mathsf{new}; x := \bar{i}; \Pi_{f \in F}\bar{i}_f := \mathsf{nil}).$$

It is worthwhile to note that this translation is based on the *reuse* of a *variable* $\bar{i} \in I$ which in fact can be seen as a *canonical representative* of those variables which refer to the same object.

Finally we consider the case of a procedure call. It is not so difficult to see that upon the return of a procedure call in the *translated* program some objects which are reachable from the restored local variables may no longer be represented by a global variable $\bar{i} \in I$ because their representation may have been *reused* by the creation of new objects. In order to restore the representation of such objects we introduce for each $\bar{i} \in I$ fresh variables $\bar{i}' \in I'$ and $\bar{i}'_f$, for $f \in F$, which are used to store *before* the call a copy of the heap (as represented by the variables in I) by the following statement

$$copy = \Pi_{\bar{i}' \in I'}(\bar{i}' := \bar{i}; \Pi_{f \in F}\bar{i}'_f := \bar{i}_f).$$

After the call we first compute which variables $\bar{i}' \in I'$ represent objects which are reachable from the (restored) local variables $L(P)$ of P in the "old" heap represented by the variables $\bar{i}' \in I'$ and $\bar{i}'_f$, for $f \in F$. Assuming for each variable $\bar{i}' \in I'$ an additional fresh variable $\bar{i}'_b$, this is computed by the statement $R'_{L(P)}$ which is obtained from $R_{L(P)}$, as defined above, by replacing simply the variables $\bar{i}$, $\bar{i}_f$ and $\bar{i}_b$ by $\bar{i}'$, $\bar{i}'_f$ and $\bar{i}'_b$, respectively, for $\bar{i} \in I$ and $f \in F$. Next we compute by $R_{G(P)}$ which variables $\bar{i} \in I$ do *not* represent objects reachable from the global variables $G(P)$ of P in the current heap. The statement

$$R_{\bar{i}'} = b := \mathsf{true}; \mathit{\Pi}_{\bar{j} \in I}([\bar{j} \neq \bar{i}']\mathit{skip} + [\bar{j} = \bar{i}']b := \mathsf{false})$$

checks whether the object denoted by $\bar{i}'$ is already represented by some variable $\bar{i} \in I$. If the object denoted by $\bar{i}'$ is not yet represented the following statement

$$\mathit{restore} = \mathit{\Sigma}_{\bar{j} \in I}([\bar{j}_b = \mathsf{false}]\bar{j} := \bar{i}'; \mathit{\Pi}_{f \in F}\bar{j}_f := \bar{i}'_f; \bar{j}_b := \mathsf{true})$$

restores the representation of $\bar{i}'$. Putting the above statements together

$$\begin{aligned} \mathit{return} = {} & R'_{L(P)}; R_{G(P)}; \\ & \mathit{\Pi}_{\bar{i}' \in I'} R_{\bar{i}'}; ([b = \mathsf{false}]\mathit{skip} + [b = \mathsf{true}]\mathit{restore}) \end{aligned}$$

restores upon return the representation of the old local heap by the variables I.

Summarizing, we have the following translation of procedure calls:

$$t(\mathsf{call}\, p) = \mathit{copy}; \mathsf{call}\, p; \mathit{return}.$$

In order to state the correctness of the translation in terms of a bisimulation relation we first extend pointwise the (representation) relation $H \equiv s$ to the corresponding stacks:

1. if $H \equiv s$ and $\Gamma \equiv S$ then $H \bullet \Gamma \equiv s \bullet S$,
2. if $\Gamma \equiv S$ then $B \bullet \Gamma \equiv t(B) \bullet S$.

Let $\Longrightarrow$ denote the concrete semantics where the translations of assignments, procedure calls and returns are executed *atomically* (i.e., in one step).

Theorem 13. *Given a program P with fields, let $H \bullet \Gamma \equiv s \bullet S$. We have*

1. *if $\langle H, \Gamma \rangle \longrightarrow \langle H', \Gamma' \rangle$ then $\langle s, S \rangle \Longrightarrow \langle s', S' \rangle$, for some $\langle s', S' \rangle$ such that $H' \bullet \Gamma' \equiv s' \bullet S'$, and*
2. *if $\langle s, S \rangle \Longrightarrow \langle s', S' \rangle$ then $\langle H, \Gamma \rangle \longrightarrow \langle H', \Gamma' \rangle$, for some $\langle H', \Gamma' \rangle$ such that $H' \bullet \Gamma' \equiv s' \bullet S'$.*

5 Conclusion

The interplay between unbounded allocation of objects and recursion with local variables gives rise to an infinite state space. By representing the state space symbolically, we have shown that reachability, as well as model checking, is

decidable. Further, we have shown that adding pointer fields to our core language with the restriction that the number of reachable objects is bounded, does not increase the expressiveness of the language.

Our symbolic semantics greatly simplifies the basic mechanism of recursion with local variables in the presence of dynamic object allocation, as is also exemplified by a neat formalization of dynamic deallocation, a feature that is typically problematic in the context of model checking. Consider for example a *deallocation* statement "$\mathsf{del}\, x$" that sets x and all of its aliases to nil. Note that this is different from an assignment $x := \mathsf{nil}$, as the latter statement does not affect any alias of x. Symbolically, the effect of a deallocation statement is formalized in terms of its strongest postcondition $SP(\mathsf{del}\, x, \varphi)$ simply as $\varphi \wedge x = \mathsf{nil}$. An equivalent concrete semantics for deallocation is much more complex because the stack may contain variables still referencing deallocated objects and a program has only access to the top of the stack. One way of implementing a concrete semantics of deallocation is by an explicit recording of the deleted objects.

Finally, the symbolic nature of our semantics provides a promising basis for future model checking tool development using, for example, the Maude implementation of rewriting logic.

References

1. Abdulla, P.A., Atig, M.F., Delzanno, G., Podelski, A.: Push-Down Automata with Gap-Order Constraints. In: Arbab, F., Sirjani, M. (eds.) FSEN 2013. LNCS, vol. 8161, pp. 191–206. Springer, Heidelberg (2013)
2. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
3. Bouajjani, A., Fratani, S., Qadeer, S.: Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220. Springer, Heidelberg (2007)
4. Corbett, J., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: Proc. of Int. Conf. on Software Engineering, pp. 439–448. IEEE (2000)
5. Demartini, C., Iosif, R., Sisto, R.: A deadlock detection tool for concurrent Java programs. Software-Practice and Experience 29(7), 577–603 (1999)
6. Distefano, D., Katoen, J.-P., Rensink, A.: Who is Pointing When to Whom? In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 250–262. Springer, Heidelberg (2004)
7. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
8. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. Int. Jour. on Softw. Tools for Technology Transfer 2(4), 366–381 (2000)
9. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall (1967)
10. Park, D., Stern, U., Skakkebaek, J., Dill, D.: Java Model Checking. In: Proc. of the 15th Int. Conf. on Automated Software Engineering, pp. 253–256. IEEE (2000)
11. Rot, J., Asavoae, I.M., de Boer, F., Bonsangue, M.M., Lucanu, D.: Interacting via the Heap in the Presence of Recursion. In: Proc. of the 5th Interaction and Concurrency Experience, ICE 2012. EPTCS, vol. 104, pp. 99–113 (2012)

On the Complexity of Adding Convergence*

Alex Klinkhamer and Ali Ebnenasir

Department of Computer Science,
Michigan Technological University,
Houghton, MI 49931, U.S.A.
{apklinkh, aebnenas}@mtu.edu

Abstract. This paper investigates the complexity of designing Self-Stabilizing (SS) distributed programs, where an SS program meets two properties, namely closure and convergence. *Convergence* requires that, from *any* state, the computations of an SS program reach a set of legitimate states (a.k.a. *invariant*). Upon reaching a legitimate state, the computations of an SS program remain in the set of legitimate states as long as no faults occur; i.e., *Closure*. We illustrate that, in general, the problem of augmenting a distributed program with convergence, i.e., *adding convergence*, is NP-complete (in the size of its state space). An implication of our NP-completeness result is the hardness of adding non-masking fault tolerance to distributed programs, which has been an open problem for the past decade.

Keywords: Self-Stabilization, Convergence, NP-Completeness

1 Introduction

Today's distributed programs are subject to a variety of transient faults due to their inherent complexity, human errors and environmental factors, where transient faults perturb program state without causing any permanent damage (e.g., bad initialization, loss of coordination, soft errors). Distributed applications should guarantee service availability even in the presence of faults. However, providing global recovery in distributed programs is difficult in part due to (1) no central point of control/administration; (2) lack of knowledge about the global state of the program by program processes/components, and (3) the need for global recovery using only the local actions of processes. To design programs that recover from any arbitrary configuration/state without human intervention, Dijkstra [1] proposed self-stabilization as a property of distributed programs. A Self-Stabilizing (SS) program meets two requirements, namely closure and convergence. *Convergence* requires that, from *any* state, the computations of an SS program reach a set of legitimate states (a.k.a. program *invariant*). Upon reaching an invariant state, the computations of an SS program remain in its

* This work was sponsored in part by the NSF grant CCF-1116546 and a grant from Michigan Technological University.

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 17–33, 2013.
DOI: 10.1007/978-3-642-40213-5_2,
© IFIP International Federation for Information Processing 2013

invariant as long as no faults occur; i.e., *Closure*. Indeed, self-stabilization is a special case of *nonmasking* fault tolerance [2,3], where instead of providing recovery from any state, designers identify a subset of the state space from where recovery to the invariant can be provided. While there are several approaches in the literature for the design of SS algorithms for specific problems, we are not aware of a general case complexity analysis of designing SS programs.

Several researchers have investigated the problem of adding nonmasking fault tolerance to programs [1,2,4,3,5,6]. For instance, Liu and Joseph [4] present a method for the transformation of an intolerant program to a fault-tolerant version thereof by going through a set of refinement steps. Arora and Gouda [2,3] use the notions of closure and convergence to define three levels of fault tolerance based on the extent to which safety and liveness specifications [7] are satisfied in the presence of faults. In their setting, a *failsafe* fault-tolerant program ensures its safety at all times even if faults occur, whereas, in the presence of faults, a *nonmasking* program provides recovery to its invariant; no guarantees on meeting safety during recovery. A *masking* fault-tolerant program is both failsafe and nonmasking. Arora *et al.* [5] design nonmasking fault tolerance by creating a dependency graph of the local constraints of program processes, and by illustrating how these constraints should be satisfied so global recovery is achieved. Kulkarni and Arora [6] demonstrate that adding failsafe/nonmasking/masking fault tolerance to high atomicity programs can be done in polynomial-time in program's state space, where a *high atomicity* program can read/write all program variables in an atomic step. Nonetheless, they illustrate that adding masking fault tolerance to *low atomicity* programs – where processes have read/write restrictions with respect to variables of other processes – is NP-complete (in the size of the state space).[1] Moreover, Kulkarni and Ebnenasir [8] show that adding failsafe fault tolerance to low atomicity programs is also an NP-complete problem. Nonetheless, while adding nonmasking fault tolerance is known to be in NP, no polynomial-time algorithms are known for efficient design of nonmasking fault tolerance for low atomicity programs; nor has there been a proof of NP-hardness!

In this paper, we illustrate that adding nonmasking fault tolerance to low atomicity programs is NP-complete (in the size of the state space). Our hardness proof is based on a reduction from the 3-SAT problem [9] to the problem of adding convergence to non-stabilizing programs. Since adding convergence is a special case of adding nonmasking fault tolerance, it follows that, in general, it is unlikely that adding nonmasking fault tolerance to low atomicity programs can be done efficiently (unless $P = NP$). The significance of our NP-hardness proof is multi-fold. First, our proof provides a solution for a problem that has been open for more than a decade. Second, illustrating the NP-completeness of adding convergence is particularly important since the proof requires the construction of the entire state space of the instance of the problem of adding convergence, yet such a mapping should be polynomial in the size of the instance of the source NP-complete problem (in our case 3-SAT). Devising such a reduction has been another open problem in the literature. Third, our proof illustrates that even if

[1] Low atomicity programs enable the modeling of distributed programs.

we have a process in a program that can atomically read the global state of the program and can update its own local state, the addition of recovery still remains a hard problem. Fourth, the presented hardness proof lays the foundation for the design of a new family of synthesis algorithms inspired by the DPLL algorithm [10], which we are currently investigating. We conjecture that such synthesis algorithms will be more efficient than existing methods of SAT-based synthesis of fault tolerance [11] where one formulates the sub-problems of adding fault tolerance in terms of CNF formulas and invokes off-the-shelf SAT solvers.

Organization. Section 2 presents the basic concepts of programs, faults and fault tolerance. Section 3 formally states the problem of adding nonmasking fault tolerance and convergence. Section 4 illustrates that adding convergence in particular and adding nonmasking fault tolerance in general are NP-complete. Section 5 discusses related work. Finally, Section 6 makes concluding remarks and discusses future work.

2 Preliminaries

In this section, we present the formal definitions of programs, faults, fault tolerance and self-stabilization, and our distribution model (adapted from [6]). Programs are defined in terms of their set of variables, their transitions, and their processes. The definitions of fault tolerance and self-stabilization are adapted from [1,3,12,13]. For ease of presentation, we use a simplified version of Dijkstra's token ring protocol [1] as a running example.

Programs as (non-deterministic) finite-state machines. A program in our setting is a representation of any system that can be captured by a (finite-state) non-deterministic state machine (e.g., network protocols). Formally, a *program* p is a tuple $\langle V_p, \delta_p, \Pi_p, T_p \rangle$ of a finite set V_p of variables, a set δ_p of transitions, a finite set Π_p of k processes, where $k \geq 1$, and a topology T_p. Each variable $v_i \in V_p$, for $i \in \mathbb{N}_m$ where $\mathbb{N}_m = \{0, 1, \cdots, m-1\}$ and $m > 0$, has a finite non-empty domain D_i. A *state* s of p is a valuation $\langle d_0, d_1, \cdots, d_{m-1} \rangle$ of variables $\langle v_0, v_1, \cdots, v_{m-1} \rangle$, where $d_i \in D_i$. A *transition* t is an ordered pair of states, denoted (s_0, s_1), where s_0 is the source and s_1 is the target/destination state of t. A *deadlock state* is a state with no outgoing transitions. For a variable v and a state s, $v(s)$ denotes the value of v in s. The *state space* of p, denoted S_p, is the set of all possible states of p, and $|S_p|$ denotes the size of S_p. A *state predicate* is any subset of S_p specified as a Boolean expression over V_p. We say a state predicate X *holds in a state* s (respectively, $s \in X$) *if and only if (iff)* X evaluates to true at s.

Read/Write model. We adopt a shared memory model [14] since reasoning in a shared memory setting is easier, and several (correctness-preserving) transformations [15,16] exist for the refinement of shared memory fault-tolerant programs to their message-passing versions. We model the topological constraints (denoted T_p) of a program p by a set of read and write restrictions imposed on variables that identify the locality of each process. Specifically, we consider a subset of variables in V_p that a process P_j ($j \in \mathbb{N}_k$) can write, denoted w_j, and

a subset of variables that P_j is allowed to read, denoted r_j. We assume that for each process P_j, $w_j \subseteq r_j$; i.e., if a process can write a variable, then it can also read that variable.

Impact of read/write restrictions. Every transition of a process P_j belongs to a *group* of transitions due to the inability of P_j to read variables that are not in r_j. Consider two processes P_0 and P_1 each having a Boolean variable that is not readable for the other process. That is, P_0 (respectively, P_1) can read and write x_0 (respectively, x_1), but cannot read x_1 (respectively, x_0). Let $\langle x_0, x_1 \rangle$ denote a state of this program. Now, if P_0 writes x_0 in a transition $(\langle 0,0\rangle, \langle 1,0\rangle)$, then P_0 has to consider the possibility of x_1 being 1 when it updates x_0 from 0 to 1. As such, executing an action in which the value of x_0 is changed from 0 to 1 is captured by the fact that a group of two transitions $(\langle 0,0\rangle, \langle 1,0\rangle)$ and $(\langle 0,1\rangle, \langle 1,1\rangle)$ is included in P_0. In general, a transition is included in the set of transitions of a process *iff* its associated group of transitions is included. Formally, any two transitions (s_0, s_1) and (s_0', s_1') in a group of transitions formed due to the read restrictions of a process P_j, denoted r_j, meet the following constraints: $\forall v : v \in r_j : (v(s_0) = v(s_0')) \wedge (v(s_1) = v(s_1'))$ and $\forall v : v \notin r_j : (v(s_0) = v(s_1)) \wedge (v(s_0') = v(s_1'))$.

Due to read/write restrictions, a *process* P_j ($j \in \mathbb{N}_k$) includes a set of transition groups $P_j = \{g_{j0}, g_{j1}, \cdots, g_{j(max-1)}\}$ created due to read restrictions r_j, where $max \geq 1$. Due to write restrictions w_j, no transition group g_{ji} ($i \in \mathbb{N}_{max}$) can have a transition (s_0, s_1) that updates a variable $v \notin w_j$. Thus, the set of transitions δ_p of a program p is equal to the union of the transition groups of its processes; i.e., $\delta_p = \cup_{j=0}^{k-1} P_j$. (It is known that the total number of groups is polynomial in $|S_p|$ [6]). We use p and δ_p interchangeably.

To simplify the specification of δ_p for designers, we use Dijkstra's guarded commands language [17] as a shorthand for representing the set of program transitions. A guarded command (action) is of the form $grd \rightarrow stmt$, and includes a set of transitions (s_0, s_1) such that the predicate grd holds in s_0 and the atomic execution of the statement $stmt$ results in state s_1. An action $grd \rightarrow stmt$ is *enabled* in a state s iff grd holds at s. A process $P_j \in \Pi_p$ is *enabled* in s iff there exists an action of P_j that is enabled at s.

Example: Token Ring (TR). The Token Ring (TR) program (adapted from [1]) includes three processes $\{P_0, P_1, P_2\}$ each with an integer variable x_j, where $j \in \mathbb{N}_3$, with a domain $\{0, 1, 2\}$. The process P_0 has the following action (addition and subtraction are in modulo 3):

$$A_0 : (x_0 = x_2) \quad \longrightarrow \quad x_0 := x_2 + 1$$

When the values of x_0 and x_2 are equal, P_0 increments x_0 by one. We use the following parametric action to represent the actions of processes P_j, for $1 \leq j \leq 2$:

$$A_j : (x_j \neq x_{(j-1)}) \quad \longrightarrow \quad x_j := x_{(j-1)}$$

Each process P_j copies x_{j-1} only if $x_j \neq x_{j-1}$, where $j = 1, 2$. By definition, process P_j *has a token* iff $x_j \neq x_{j-1}$. Process P_0 *has a token* iff $x_0 = x_2$. We define a state predicate I_{TR} that captures the set of states in which only one token exists, where I_{TR} is

$$((x_0 = x_1) \wedge (x_1 = x_2)) \ \vee \ ((x_1 \neq x_0) \wedge (x_1 = x_2)) \ \vee \ ((x_0 = x_1) \wedge (x_1 \neq x_2))$$

Each process P_j $(1 \leq j \leq 2)$ is allowed to read variables x_{j-1} and x_j but can write only x_j. Process P_0 is permitted to read x_2 and x_0 and can write only x_0. Thus, since a process P_j is unable to read one variable (with a domain of three values), each group associated with an action A_j includes three transitions. For a TR protocol with n processes and with n values in the domain of each variable x_j, each group includes n^{n-2} transitions. ◁

Computations. Intuitively, a computation of a program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$ is an *interleaving* of its actions. Formally, a *computation* of p is a sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states that satisfies the following conditions: (1) for each transition (s_i, s_{i+1}) in σ, where $i \geq 0$, there exists an action $grd \rightarrow stmt$ in some process $P_j \in \Pi_p$ such that grd holds at s_i and the execution of $stmt$ at s_i yields s_{i+1}, and (2) σ is *maximal* in that either σ is infinite or if it is finite, then σ reaches a state s_f where no action is enabled. A *computation prefix* of a program p is a *finite* sequence $\sigma = \langle s_0, s_1, \cdots, s_z \rangle$ of states, where $z > 0$, such that each transition (s_i, s_{i+1}) in σ $(i \in \mathbb{N}_z)$ belongs to some action $grd \rightarrow stmt$ in some process $P_j \in \Pi_p$. The *projection* of a program p on a non-empty state predicate X, denoted as $\delta_p | X$, is the program $\langle V_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \wedge s_0, s_1 \in X\}, \Pi_p, T_p \rangle$. In other words, $\delta_p | X$ consists of transitions of p that start in X and end in X.

Closure and invariant. A state predicate X is *closed in an action* $grd \rightarrow stmt$ *iff* executing $stmt$ from any state $s \in (X \wedge grd)$ results in a state in X. We say a state predicate X is *closed in a program* p *iff* X is closed in every action of p. In other words, *closure* [13] requires that every computation of p starting in X remains in X. We say a state predicate I is an *invariant* of p *iff* I is closed in p.

TR Example. Starting from a state in the predicate I_{TR}, the TR protocol generates an infinite sequence of states, where all reached states belong to I_{TR}. ◁

Faults. Intuitively, we capture the impact of faults on a program as state perturbations. Formally, a class of *faults* f for a program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$ is a subset of $S_p \times S_p$. We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in δ_p and the transitions in f. We say that a state predicate T is an f-span (read as *fault-span*) of p from a state predicate I *iff* the following two conditions are satisfied: (1) $I \subseteq T$, and (2) T is closed in $p[]f$. Observe that for all computations of p that start in I, the state predicate T is a superset of I in S_p up to which the state of p may be perturbed by the occurrence of f transitions. We say that a sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$ is a *computation of* p *in the presence of* f *iff* the following conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in (p[]f)$; (2) if σ is finite and terminates in state s_l, then there is no state s such that $(s_l, s) \in \delta_p$, and (3) $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_p)$. The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute. That is, if the only transition that starts from s_l is a fault transition (s_l, s_f) then as far as the program is concerned, s_l is still a deadlock state because the program does not have control over the execution of (s_l, s_f); i.e., (s_l, s_f) may or

may not be executed. Finally, the third requirement captures that the number of fault occurrences in a computation is finite. This requirement is the same as that made in previous work (e.g., [1,18,3,19]) to ensure that eventually recovery can occur. In the same way that we use guarded commands to represent program transitions, we use them to specify fault transitions. That is, the impact of faults can be captured as a set of actions that update program variables.
TR Example. The TR protocol is subject to transient faults that can perturb its state to an arbitrary state. For instance, the following action captures the impact of faults on x_0, where | denotes non-deterministic assignment of values to x_0:

$$F_0 : = \text{true} \quad \longrightarrow \quad x_0 := 0|1|2;$$

The impact of faults on x_1 and x_2 are captured with two actions F_1 and F_2 symmetric to F_0. $\triangleleft$
Nonmasking fault tolerance and self-stabilization. Let I be a state predicate closed in a program p and f be a class of faults. We say that p is nonmasking f-tolerant from I *iff* there exists an f-span of p from I, denoted T, such that T *converges* to I in p. That is, from any state $s_0 \in T$, *every* computation of p that starts in s_0 reaches a state where I holds. We say that p is *self-stabilizing* from I *iff* p is nonmasking f-tolerant from I, where $T = \mathit{true}$. That is, the f-span of p is equal to S_p, and convergence to I is guaranteed from any state in S_p. Notice that, to design recovery, one has to ensure that no deadlock states exist in $T-I$, and no non-progress cycles exist in $\delta_p \mid (T-I)$. A *non-progress cycle* (a.k.a. *livelock*) in $\delta_p \mid (T-I)$ is a sequence of states $\sigma = \langle s_0, s_1, \cdots, s_m, s_0 \rangle$, where $m \geq 0$, $(s_i, s_{i\oplus 1}) \in \delta_p$ and $s_i \in (T-I)$, for $i \in \mathbb{N}_{m+1}$ and $\oplus$ denotes addition modulo $m+1$.

Proposition 1. *A program p is nonmasking f-tolerant from I with a f-span T iff there are no deadlock states in $T-I$ and no non-progress cycles in $\delta_p \mid (T-I)$.*

Note. In this paper, we analyze the complexity of adding convergence under the assumption of no fairness.

3 Problem Statement

In this section, we represent the problem of adding nonmasking fault-tolerance from [6]. Consider a fault-intolerant program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$, a class of faults f, and a state predicate I, where I is closed in p. Our objective is to generate a revised version of p, denoted p', such that p' is nonmasking f-tolerant from an invariant I'. To separate fault tolerance from functional concerns, we would like to preserve the behaviors of p in the absence of f during the addition of fault tolerance. For this reason, during the synthesis of p' from p, no states (respectively, transitions) are added to I (respectively, $\delta_p|I$). Thus, we have $I' \subseteq I$ and $p'|I' \subseteq p|I'$. This way, if p' starts in I' in the absence of faults, then p' will preserve the correctness of p; i.e., the added recovery does not interfere with normal functionalities of p in the absence of faults. Moreover, if p' starts in a state outside I', then only recovery to I' will be provided by p'. Thus, we formally state the problem as follows:

Problem 1. **Adding Nonmasking Fault Tolerance**

- **Input**: (1) A program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$; (2) A class of faults f; (3) A state predicate I such that I is closed in p, and (4) topological constraints of p captured by read/write restrictions.
- **Output**: A program $p' = \langle V_{p'}, \delta_{p'}, \Pi_{p'}, T_{p'} \rangle$ and a state predicate I' such that the following constraints are met: (1) I' is non-empty and $I' \subseteq I$; (2) $\delta_{p'}|I' \subseteq \delta_p|I'$; (3) Π_p and $\Pi_{p'}$ have the same number of processes and $T_p = T_{p'}$, and (4) p' is nonmasking f-tolerant from I'. □

We state the corresponding decision problem as follows:

Problem 2. **Decision Problem of Adding Nonmasking Fault Tolerance**

- INSTANCE: (1) A program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$; (2) A class of faults f; (3) A state predicate I such that I is closed in p, and (4) topological constraints of p captured by read/write restrictions.
- QUESTION: Does there exist a program $p' = \langle V_{p'}, \delta_{p'}, \Pi_{p'}, T_{p'} \rangle$ and a state predicate I' such that the constraints of Problem 1 are met? □

A special case of Problem 2 is where f denotes a class of transient faults, $I = I'$, $\delta_{p'}|I' = \delta_p|I'$, and p' is self-stabilizing to I.

Problem 3. **Decision Problem of Adding Convergence**

- INSTANCE: (1) A program $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$; (2) A state predicate I such that I is closed in p, and (3) topological constraints captured by read/write restrictions.
- QUESTION: Does there exist a program p_{ss} with an invariant I_{ss} such that $I = I_{ss}$, $\delta_{p_{ss}}|I_{ss} = \delta_p|I_{ss}$, and p_{ss} is self-stabilizing to I_{ss}? □

4 Hardness Results

In this section, we illustrate that adding convergence to low atomicity programs is NP-complete (in the size of the state space). This hardness result implies the hardness of general case addition of nonmasking fault tolerance to low atomicity programs (i.e., Problem 2). Specifically, we demonstrate that, for a given intolerant program p with an invariant I, adding convergence from S_p to I is an NP-hard problem. Section 4.1 presents a polynomial-time mapping from 3-SAT to an instance of Problem 3. Section 4.2 shows that the instance of 3-SAT is satisfiable *iff* a self-stabilizing version of the instance of Problem 3 exists.

Problem 4. The 3-SAT decision problem.

- INSTANCE: A set $\mathcal{V}$ of n propositional variables $(v_0, \cdots, v_{n-1})$ and k clauses $(C_0, \cdots, C_{k-1})$ over $\mathcal{V}$ such that each clause is of the form $(l_q \vee l_r \vee l_s)$, where $q, r, s \in \mathbb{N}_k$ and $\mathbb{N}_k = \{0, 1, \cdots, k-1\}$. Each l_r denotes a literal, where a literal is either $\neg v_r$ or v_r for $v_r \in \mathcal{V}$. We assume that $\neg(q = r = s)$ holds for all clauses; otherwise, the 3-SAT instance can efficiently be transformed to a formula that meets this constraint.

– QUESTION: Is there a satisfying truth-value assignment for the variables in $\mathcal{V}$ such that each C_i evaluates to *true*, for all $i \in \mathbb{N}_k$?

Notation. We say l_r is a *negative* (respectively, *positive*) literal *iff* it has the form $\neg v_r$ (respectively, v_r), where $v_r \in \mathcal{V}$. Consider a clause $C_i = (l_q \vee l_r \vee l_s)$. We use a binary variable b^i_j, where $i \in \mathbb{N}_k$ and $j \in \mathbb{N}_3$, to denote the sign of the first, second, and the third literal in C_i. For example, if $l_q = \neg v_q, l_r = v_r$ and $l_s = \neg v_s$, then we have $b^i_0 = 0, b^i_1 = 1$ and $b^i_2 = 0$. Accordingly, for each clause C_i, we define a tuple $B^i = \langle b^i_0, b^i_1, b^i_2 \rangle$. Notice that, the binary variable b^i_j is independent from the indices of the literals in clause C_i and represents only the positive/negative form of the three literals in C_i.

4.1 Polynomial Mapping

In this section, we present a polynomial-time mapping from an instance of 3-SAT to the instance of Problem 3, denoted $p = \langle V_p, \delta_p, \Pi_p, T_p \rangle$. That is, corresponding to each propositional variable and clause, we illustrate how we construct a program p, its processes Π_p, its variables V_p, its read/write restrictions and its invariant I. We shall use this mapping in Section 4.2 to demonstrate that the instance of 3-SAT is satisfiable *iff* convergence from S_p can be added to p.

Processes, variables, and read/write restrictions. We consider three processes, P_0, P_1, and P_2 in p. Each process P_j ($j \in \mathbb{N}_3$) has two variables x_j and y_j, where the domain of x_j is equal to $\mathbb{N}_n = \{0, 1, \cdots, n-1\}$ and y_j is a binary variable. (Notice that n denotes the number of propositional variables in the 3-SAT instance.) The process P_j can read both x_j and y_j but can write only y_j. We also consider a fourth process P_3 that can read all variables and write a binary variable $sat \in \mathbb{N}_2$. The variable sat can be read by processes P_0, P_1, and P_2, but not written. That is, we have $r_j = \{x_j, y_j, sat\}$, and $w_j = \{y_j\}$ for

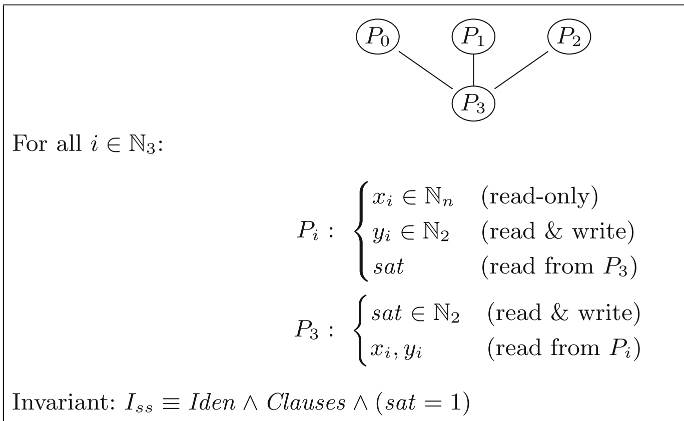


Fig. 1. Instance of Problem 3

$j \in \mathbb{N}_3$, and $r_3 = \{x_0, y_0, x_1, y_1, x_2, y_2, sat\}$ and $w_3 = \{sat\}$ (see Figure 1). We also have $V_p = \{x_0, y_0, x_1, y_1, x_2, y_2, sat\}$, and $\Pi_p = \{P_0, P_1, P_2, P_3\}$.

Invariant/legitimate states. Inspired by the form of the 3-SAT instance and its requirements, we define a state predicate I_{ss} that denotes the invariant of p.

- Corresponding to each clause $C_i = (l_q \vee l_r \vee l_s)$, we construct a state predicate $PredC_i \equiv (x_0 = q \implies y_0 = b_0^i) \vee (x_1 = r \implies y_1 = b_1^i) \vee (x_2 = s \implies y_2 = b_2^i)$. In other words, we have $PredC_i \equiv ((x_0 = q) \wedge (x_1 = r) \wedge (x_2 = s)) \implies ((y_0 = b_0^i) \vee (y_1 = b_1^i) \vee (y_2 = b_2^i))$. This way, we construct a state predicate $Clauses \equiv (\forall i \in \mathbb{N}_k : PredC_i)$. Notice that we check the value of each x_j with respect to the index of the literal appearing in position j in C_i, where $j \in \mathbb{N}_3$. This is due to the fact that the domain of x_j is equal to the range of the indices of propositional variables (i.e., $\mathbb{N}_n$).
- A literal l_r may appear in positions i and j in distinct clauses of 3-SAT, where $i, j \in \mathbb{N}_3$ and $i \neq j$. Since each propositional variable $v_r \in \mathcal{V}$ gets a unique truth-value in 3-SAT, the truth-value of l_r is independent from its position in the 3-SAT formula. Given the way we construct the state predicate *Clauses*, it follows that, in the instance of Problem 3, whenever $x_i = x_j$ we should have $y_i = y_j$. Thus, we construct the state predicate $Iden \equiv (\forall i, j \in \mathbb{N}_3 : (x_i = x_j \implies y_i = y_j))$, which is conjoined with the predicate *Clauses*.
- In the instance of Problem 3, we require that $(sat = 1)$ holds in all invariant states.

Thus, the invariant of p is equal to the state predicate I_{ss}, where

$$I_{ss} \equiv Iden \wedge Clauses \wedge (sat = 1)$$

Notice that, the size of the state space of p is equal to $2\,(2n)^3$; i.e., $|S_p|$ is polynomial in the size of the 3-SAT instance.

Example 1. Example Construction

Let us consider the 3-SAT formula $\phi \equiv (v_0 \vee v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_1 \vee v_2) \wedge (v_1 \vee \neg v_2 \vee \neg v_0)$. Since there are three propositional variables and four clauses, we have $n = 3$ and $k = 4$. Moreover, based on the mapping described before, we have $C_0 \equiv (v_0 \vee v_1 \vee v_2)$, $C_1 \equiv (\neg v_1 \vee \neg v_1 \vee \neg v_2)$, $C_2 \equiv (\neg v_1 \vee \neg v_1 \vee v_2)$ and $C_3 \equiv (v_1 \vee \neg v_2 \vee \neg v_0)$. We have $B^0 = \langle 1, 1, 1\rangle$, $B^1 = \langle 0, 0, 0\rangle$, $B^2 = \langle 0, 0, 1\rangle$ and $B^3 = \langle 1, 0, 0\rangle$. The state predicate $Iden$ is defined as before and $Clauses$ is the conjunction of each $PredC_i$ ($i \in \mathbb{N}_4$) defined as follows:

$$PredC_0 \equiv (x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 1 \vee y_1 = 1 \vee y_2 = 1)$$
$$PredC_1 \equiv (x_0 = 1 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 0 \vee y_1 = 0 \vee y_2 = 0)$$
$$PredC_2 \equiv (x_0 = 1 \wedge x_1 = 1 \wedge x_2 = 2) \implies (y_0 = 0 \vee y_1 = 0 \vee y_2 = 1)$$
$$PredC_3 \equiv (x_0 = 1 \wedge x_1 = 2 \wedge x_2 = 0) \implies (y_0 = 1 \vee y_1 = 0 \vee y_2 = 0)$$

4.2 Correctness of Reduction

In this section, we illustrate that Problem 3 is NP-complete. Specifically, we show that the instance of 3-SAT is satisfiable *iff* convergence from S_p to I_{ss} can be added to the instance of Problem 3, denoted p.

Lemma 1. *If the instance of 3-SAT has a satisfying valuation, then convergence from S_p can be added to the instance of Problem 3; i.e., there is a self-stabilizing version of p, denoted p_{ss}.*

Let there be a truth-value assignment to the propositional variables in $\mathcal{V}$ such that every clause evaluates to *true*; i.e., $\forall i : i \in \mathbb{N}_k : C_i$. Initially, $\delta_p = \emptyset$ and $\delta_p = \delta_{p_{ss}}$. Based on the value assignments to propositional variables, we include a set of transitions (represented as convergence actions) in p_{ss}. Then, we illustrate that the following three properties hold: the invariant $I_{ss} \equiv Clauses \wedge Iden \wedge (sat = 1)$ remains closed in p_{ss}, deadlock-freedom in $\neg I_{ss}$, and livelock-freedom in $p_{ss}|\neg I_{ss}$.

- If a propositional variable v_r (where $r \in \mathbb{N}_n$) is assigned *true*, then we include the following action in each process P_j, where $j \in \mathbb{N}_3$: $x_j = r \wedge y_j = 0 \wedge sat = 0 \rightarrow y_j := 1$.
- If a propositional variable v_r (where $r \in \mathbb{N}_n$) is assigned *false*, then we include the following action in each process P_j, where $j \in \mathbb{N}_3$: $x_j = r \wedge y_j = 1 \wedge sat = 0 \rightarrow y_j := 0$.
- We include the following actions in P_3: $(Iden \wedge Clauses) \wedge sat = 0 \rightarrow sat := 1$ and $\neg\,(Iden \wedge Clauses) \wedge sat = 1 \rightarrow sat := 0$.

Now we illustrate that closure, deadlock-freedom, and livelock-freedom hold. That is, the resulting program is self-stabilizing to I_{ss}.

Closure. Since the first three processes can execute an action only in states where $sat = 0$, their actions are disabled where $sat = 1$. Thus, the first three processes exclude any transition that starts in I_{ss}; i.e., preserving the closure of I_{ss} and ensuring $p_{ss}|I_{ss} \subseteq p|I_{ss}$. Moreover, P_3 takes an action only when its guards are enabled; i.e., in illegitimate states. Therefore, none of the included actions violate the closure of I_{ss}, and the second constraint of the output of Problem 1 holds.

Livelock Freedom. To show livelock-freedom, we illustrate that the included actions have no circular dependencies. That is, no set of actions can enable each other in a cyclic fashion. Due to read/write restrictions, none of the three processes P_0, P_1, and P_2 executes based on the local variables of another process. Moreover, each process can update only its own y value. Once any one of the processes P_0, P_1, and P_2 updates its y value, it disables itself. Thus, the actions of one process cannot enable/disable another process. Moreover, since each action disables itself, there are no self-loops either. The guards of the actions of P_3 cannot be simultaneously *true*. Moreover, once one of them is enabled, the other one is certainly disabled, and the execution of one cannot enable another (because they only update the value of *sat*). Only processes P_0, P_1, and P_2 can make the predicate $(Iden \wedge Clauses)$ *true* when $sat = 0$. Once P_3 sets *sat* to 1 from states

$(Iden \wedge Clauses) \wedge (sat = 0)$, a state in I_{ss} is reached. Therefore, there are no cycles that start in $\neg I_{ss}$ and exclude any state in I_{ss}.

Deadlock Freedom. We illustrate that, in every state in $\neg I_{ss} \equiv (\neg(Iden \wedge Clauses) \vee (sat = 0))$, there is at least one action that is enabled.

- **Case 1**: $((Iden \wedge Clauses) \wedge (sat = 0))$ holds. In these states, the first action of P_3 is enabled. Thus, there are no deadlocks in this case.
- **Case 2**: $(\neg(Iden \wedge Clauses) \wedge (sat = 1))$ holds. In this case, the second action of P_3 is enabled. Thus, there are no deadlocks in this case.
- **Case 3**: $(\neg(Iden \wedge Clauses) \wedge (sat = 0))$ holds. None of the actions of P_3 are enabled in this case. Nonetheless, since $\neg(Iden \wedge Clauses)$ holds, either $\neg Iden$ or $\neg Clauses$, or both are *true*. When $\neg Clauses$ holds, there must be some state predicate $PredC_i$ ($i \in \mathbb{N}_k$) that is *false*. (Recall that, the invariant I_{ss} includes a state predicate $PredC_i \equiv (x_0 = q \implies y_0 = b_0^i) \vee (x_1 = r \implies y_1 = b_1^i) \vee (x_2 = s \implies y_2 = b_2^i)$ corresponding to each clause $C_i \equiv (l_q \vee l_r \vee l_s)$ in the instance of 3-SAT.) This means that the following three state predicates are *false*: $(x_0 = q \implies y_0 = b_0^i), (x_1 = r \implies y_1 = b_1^i)$ and $(x_2 = s \implies y_2 = b_2^i)$. Since the instance of 3-SAT is satisfiable, at least one of the literals l_q, l_r, or l_s must be true. As a result, based on the way we have included the actions depending on the truth-values of the propositional variables, at least one of the following actions must have been included in p_{ss}: $(x_0 = q \wedge y_0 \neq b_0^i \wedge sat = 0) \rightarrow y_0 := b_0^i$, $(x_1 = r \wedge y_1 \neq b_1^i \wedge sat = 0) \rightarrow y_1 := b_1^i$, and $(x_2 = s \wedge y_2 \neq b_2^i \wedge sat = 0) \rightarrow y_2 := b_2^i$. Thus, there is some action that is enabled when $\neg Clauses$ holds. A similar reasoning implies that there exists some action that is enabled when $\neg Iden$ holds. Thus, there are no deadlocks in Case 3.

Based on the closure of the invariant I_{ss} in all actions, deadlock-freedom in $\neg I_{ss}$, and lack of non-progress cycles in $p_{ss} | \neg I_{ss}$, it follows that the resulting program p_{ss} is self-stabilizing to I_{ss}. □

Example 2. Example Construction

In the example discussed in this section, the formula ϕ has a satisfying assignment for $v_0 = 1$, $v_1 = 0$, $v_2 = 0$. Using this value assignment, we include the following actions in the first three processes P_r where $r \in \mathbb{N}_3$:

$$x_r = 0 \wedge y_r = 0 \wedge sat = 0 \rightarrow y_r := 1$$
$$x_r = 1 \wedge y_r = 1 \wedge sat = 0 \rightarrow y_r := 0$$
$$x_r = 2 \wedge y_r = 1 \wedge sat = 0 \rightarrow y_r := 0$$

The actions of P_3 are as follows:

$$(Iden \wedge Clauses) \wedge sat = 0 \rightarrow sat := 1$$
$$\neg(Iden \wedge Clauses) \wedge sat = 1 \rightarrow sat := 0$$

Figure 2 illustrates the transitions of the stabilizing program p_{ss} originating in the state predicate $(x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2)$. Each state is represented

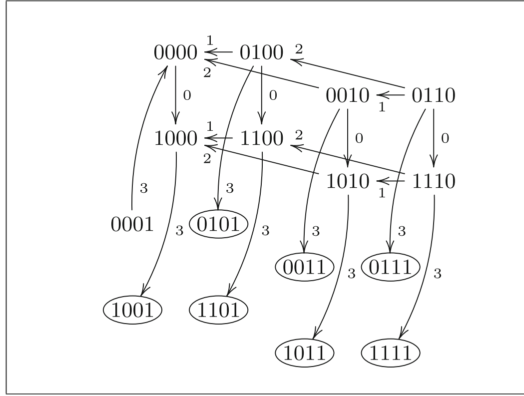


Fig. 2. Transitions originating in the state predicate $x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2$

by four bits which signify the respective values of y_0, y_1, y_2, sat. Invariant states are depicted by ovals, and the label on each transition denotes the executing process.

Lemma 2. *If there is a self-stabilizing version of the instance of Problem 3, then the corresponding 3-SAT instance has a satisfying valuation.*

By assumption, we consider a program p_{ss} to be a self-stabilizing version of p from I_{ss}. That is, p_{ss} satisfies the requirements of Problem 3.

Only P_3 can correct $(sat = 0)$. Clearly, p_{ss} must preserve the closure of I_{ss}, and should not have any deadlocks or livelocks in the states in $\neg I_{ss} \equiv (\neg(Iden \wedge Clauses) \vee (sat = 0))$. Thus, p_{ss} must include actions that correct $\neg(Iden \wedge Clauses)$ and $(sat = 0)$. Since p_{ss} must adhere to the read/write restrictions of p, only P_3 can correct $(sat = 0)$ to $(sat = 1)$. For the same reason, P_3 cannot contribute to correcting $\neg(Iden \wedge Clauses)$; only P_0, P_1, and P_2 have the write permissions to do so by updating their own y values.

The rest of the reasoning is as follows: We first illustrate that P_0, P_1, and P_2 in p_{ss} must not execute in states where $(sat = 1)$. Then, we draw a correspondence between actions included in p_{ss} and how propositional variables get unique truth-values in 3-SAT and how the clauses are satisfied.

P_0, P_1, and P_2 can be enabled only when $(sat = 0)$. We observe that no process P_j $(j \in \mathbb{N}_3)$ can have a transition that starts in the invariant I_{ss}; otherwise, the constraint $\delta_{p_{ss}}|I_{ss} \subseteq \delta_p|I_{ss}$ would be violated. We also show that no recovery action of P_0, P_1, and P_2 can include a transition that starts in a state where $sat = 1$. By contradiction, assume that some P_j $(j \in \mathbb{N}_3)$ includes a transition (s_0, s_1) where $s_0 \in \neg I_{ss}$ and $sat(s_0) = 1$ for some fixed values of x_j and y_j. Since P_j cannot read x_i and y_i of other processes P_i, where $(i \in \mathbb{N}_3) \wedge (i \neq j)$, the transition (s_0, s_1) has a groupmate (s'_0, s'_1), where $x_i(s'_0) = x_j(s'_0)$ and $y_i(s'_0) = y_j(s'_0)$ for all $i \in \mathbb{N}_3$ where $(i \neq j)$. Thus, $Iden$ is true at s'_0. Moreover, due to the form of the 3-SAT instance, no clause $(l_q \vee l_r \vee l_s)$ exists such that

$(q = r = s)$. Thus, $Clauses$ holds at s_0' as well, thereby making s_0' an invariant state. As a result, (s_0, s_1) is grouped with a transition that starts in I_{ss}, which again violates the constraint $\delta_{p_{ss}}|I_{ss} \subseteq \delta_p|I_{ss}$. Hence, P_0, P_1, and P_2 can be enabled only when $(sat = 0)$.

Actions of P_3. We show that P_3 must set sat to 0 when $\neg(Iden \wedge Clauses) \wedge sat = 1$ and may only assign sat to 1 when $(Iden \wedge Clauses) \wedge sat = 0$. As shown above, P_0, P_1, and P_2 cannot act when $sat = 1$, forcing P_3 to execute from $\neg(Iden \wedge Clauses) \wedge (sat = 1)$. P_3 must therefore have the action $\neg(Iden \wedge Clauses) \wedge sat = 1 \rightarrow sat := 0$. Consequently, P_3 cannot assign sat to 1 when $\neg(Iden \wedge Clauses) \wedge sat = 0$; otherwise, it would create a livelock with the previous action. From states where $(Iden \wedge Clauses) \wedge sat = 0$ holds, P_3 is the only process which can change sat to 1, thereby reaching an invariant state. Thus, P_3 must include the actions $\neg(Iden \wedge Clauses) \wedge sat = 1 \rightarrow sat := 0$ and $(Iden \wedge Clauses) \wedge sat = 0 \rightarrow sat := 1$.

Each P_j, for $j \in \mathbb{N}_3$ must have exactly one action for each unique value of x_j. When $sat = 0$, fixing the value of x_j to some $a \in \mathbb{N}_n$ reduces the possible local states for process P_j to 2, where $y_j = 0$ or $y_j = 1$ for $j \in \mathbb{N}_3$. (Notice that both of these states are illegitimate since $sat = 0$.) Thus, when $(x_j = a \wedge sat = 0)$ holds, process P_j has 4 possible actions: $y_j = 0 \rightarrow y_j := 0$, $y_j = 0 \rightarrow y_j := 1$, $y_j = 1 \rightarrow y_j := 0$, and $y_j = 1 \rightarrow y_j := 1$. It is clear that the first and last of those actions are self-loops and cannot be included. Thus, P_j can have either action $y_j = 0 \rightarrow y_j := 1$ or $y_j = 1 \rightarrow y_j := 0$, but not both without creating a livelock. That is, P_j cannot have more than 1 action. Additionally, to make $Iden$ true, P_j must include some action. By contradiction, assume that P_j has no actions. Another process P_i ($i \in \mathbb{N}_3$, $i \neq j$) can have $x_i = x_j$ in a non-invariant state. There are two possibilities for the y values in this state, $y_j = 0 \wedge y_i = 1$ or $y_j = 1 \wedge y_i = 0$. P_i can resolve either scenario with an action but cannot resolve both as this would require 2 actions. That is, to resolve both cases P_i needs the cooperation of P_j. Thus, P_j must have some action. Since P_j cannot have more than one action, it follows that P_j has exactly one action.

Truth-value assignment to propositional variables. Based on the above reasoning, for each value $a \in \mathbb{N}_n$, if a process P_j includes the action $x_j = a \wedge y_j = 0 \wedge sat = 0 \rightarrow y_j := 1$, then we assign *true* to the propositional variable v_a. If P_j includes the action $x_j = a \wedge y_j = 1 \wedge sat = 0 \rightarrow y_j := 0$, then we assign *false* to v_a. Let P_j include the action $x_j = a \wedge y_j = 0 \wedge sat = 0 \rightarrow y_j := 1$. By contradiction, if another process P_i, where $i \in \mathbb{N}_3 \wedge i \neq j$, includes the action $x_i = a \wedge y_i = 1 \wedge sat = 0 \rightarrow y_i := 0$, then $Iden$ would be violated and p_{ss} would never recover from the state $x_j = a \wedge x_i = a \wedge y_j = 1 \wedge y_i = 0 \wedge sat = 0$; i.e., a deadlock state, which is a contradiction with p_{ss} being self-stabilizing. Thus, each propositional variable gets a unique truth-value assignment and these value assignments are logically consistent.

Satisfying the clauses. Since p_{ss} is self-stabilizing from I_{ss}, eventually I_{ss} becomes *true*. This means that every $PredC_i$ in the $Clauses$ predicate becomes *true*. The one-to-one correspondence created by the mapping between each state predicate $PredC_i$ and each clause C_i implies that *$PredC_i$ holds iff at least*

one literal in C_i holds. Therefore, all clauses are satisfied with the truth-value assignment based on the actions of p_{ss}. □

Theorem 1. *Adding convergence to low atomicity programs is NP-complete.*

Proof. The NP-hardness of adding convergence follows from Lemmas 1 and 2. The NP membership of adding convergence has already been established in [6]; hence the NP-completeness. □

Corollary 1. *Adding nonmasking fault tolerance to low atomicity programs is NP-complete.*

Corollary 1 follows from Theorem 1 and the fact that Problem 3 is a special case of Problem 2.

5 Discussion

This section discusses extant work in three most related categories: algorithmic design of fault tolerance in general, algorithmic design of self-stabilization in particular, and complexity of algorithmic design. Several researchers have investigated the problem of algorithmic design of fault-tolerant systems [6,20,21,22,23], where a specific level of fault tolerance (e.g., failsafe, nonmasking or masking) is systematically incorporated in an existing program. Kulkarni and Arora [6] present a family of polynomial-time algorithms for the addition of different levels of fault tolerance to high atomicity programs, while demonstrating that adding masking fault tolerance to low atomicity programs is NP-complete. In our previous work [20], we establish a foundation for the addition of fault tolerance to low atomicity programs using efficient heuristics and component-based methods. Jhumka *et. al* [21,22] investigate the addition of failsafe fault tolerance under efficiency constraints. Bonakdarpour and Kulkarni [23] exploit symbolic techniques to increase the scalability of the addition of fault tolerance.

Existing methods for the algorithmic design of convergence include constraint-based methods [24] and sound heuristics [25,26]. Abujarad and Kulkarni [24] consider the program invariant as a conjunction of a set of local constraints, each representing the set of local legitimate states of a process. Then, they synthesize convergence actions for correcting the local constraints without corrupting the constraints of neighboring processes. Nonetheless, they do not explicitly address cases where local constraints have cyclic dependencies (e.g., maximal matching on a ring), and their case studies include only acyclic topologies. In our previous work [25,26], we present a method where we partition the state space to a hierarchy of state predicates based on the length of the shortest computation prefix from each state to some state in the invariant. Then, we systematically explore the space of all candidate recovery transitions that could contribute in recovery to the invariant without creating non-progress cycles outside the invariant.

Most hardness results [6,8,27] presented for the addition of fault tolerance lack the additional constraint of *recovery from any state*, which we have in the addition of convergence. The proof of NP-hardness of adding failsafe fault tolerance presented in [8] is based on a reduction from 3-SAT, nonetheless, a failsafe fault-tolerant program does not need to recover to its invariant when faults occur.

While a masking fault-tolerant program is required to recover to its invariant in the presence of faults, the problem of adding masking fault tolerance relies on finding a subset of the state space from where such recovery is possible; no need to provide recovery from any state. As such, the hardness proof presented in [6] is based on a reduction in which such a subset of state space is identified along with corresponding convergence actions if and only if the instance of 3-SAT is satisfiable. This means that some states are allowed to be excluded from the fault span; this is not an option in the case of adding convergence. The essence of the proof in [27] also relies on the same principle where Bonakdarpour and Kulkarni illustrate the NP-hardness of designing progress from one state predicate to another for low atomicity programs.

6 Conclusions and Future Work

This paper illustrates that adding convergence to low atomicity programs is an NP-complete problem, where convergence guarantees that from any state program computations recover to a set of legitimate states; i.e., invariant. In other words, we demonstrated that designing self-stabilizing programs from their non-stabilizing versions is NP-complete in the size of the state space. Since self-stabilization is a special case of nonmasking fault tolerance, it follows that adding nonmasking fault tolerance to intolerant distributed programs is also NP-complete. When faults occur, a nonmasking program guarantees recovery from states reached due to the occurrence of faults to its invariant. In the absence of faults, the computations of a nonmasking program remain in its invariant. Thus, this paper solves a decade-old open problem [6]. As an extension of this work, we will investigate special cases where the addition of convergence/nonmasking can be performed efficiently. That is, *for what programs, classes of faults and invariants the addition of convergence/nonmasking can be done in polynomial time*? Moreover, while we analyzed the general case complexity of adding convergence/nonmasking tolerance under no fairness assumption, it would be interesting to investigate the impact of different fairness assumptions on the complexity of adding convergence.

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
2. Arora, A.: A foundation of fault-tolerant computing. PhD thesis, The University of Texas at Austin (1992)
3. Arora, A., Gouda, M.G.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering 19(11), 1015–1027 (1993)
4. Liu, Z., Joseph, M.: Transformation of programs for fault-tolerance. Formal Aspects of Computing 4(5), 442–469 (1992)
5. Arora, A., Gouda, M., Varghese, G.: Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. Journal of High Speed Networks 5(3), 293–306 (1996)

6. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 82–93. Springer, London (2000)
7. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21, 181–185 (1985)
8. Kulkarni, S.S., Ebnenasir, A.: Complexity issues in automated synthesis of failsafe fault-tolerance. IEEE Transactions on Dependable and Secure Computing (TDSC) 2(3), 201–215 (2005)
9. Gary, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman (1979)
10. Davis, M., Putnam, H.: A computing procedure for quantification theory. Communications of the ACM 7(3), 201–215 (1960)
11. Ebnenasir, A., Kulkarni, S.S.: SAT-based synthesis of fault-tolerance. In: Fast Abstracts of the International Conference on Dependable Systems and Networks (2004)
12. Gouda, M.: The triumph and tribulation of system stabilization. In: Helary, J.-M., Raynal, M. (eds.) WDAG 1995. LNCS, vol. 972, pp. 1–18. Springer, Heidelberg (1995)
13. Gouda, M.G.: The theory of weak stabilization. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 114–123. Springer, Heidelberg (2001)
14. Lamport, L., Lynch, N. : Handbook of Theoretical Computer Science: Chapter 18, Distributed Computing: Models and Methods. Elsevier Science Publishers B. V. (1990)
15. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. Journal of Parallel and Distributed Computing 62(5), 766–791 (2002)
16. Demirbas, M., Arora, A.: Convergence refinement. In: Proceedings of the 22nd International Conference on Distributed Computing Systems, pp. 589–597. IEEE Computer Society, Washington, DC (2002)
17. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1990)
18. Arora, A., Kulkarni, S.S.: Designing masking fault-tolerance via nonmasking fault-tolerance. IEEE Transactions on Software Engineering 24(6), 435–450 (1998)
19. Varghese, G.: Self-stabilization by local checking and correction. PhD thesis, MIT/LCS/TR-583 (1993)
20. Ebnenasir, A.: Automatic Synthesis of Fault-Tolerance. PhD thesis, Michigan State University (May 2005)
21. Jhumka, A.: Automated design of efficient fail-safe fault tolerance. PhD thesis, Darmstadt University of Technology (2004)
22. Jhumka, A., Freiling, F.C., Fetzer, C., Suri, N.: An approach to synthesise safe systems. International Journal of Security and Networks 1(1/2), 62–74 (2006)
23. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In: Proceedings of the 27th International Conference on Distributed Computing Systems, pp. 3–10. IEEE Computer Society, Washington, DC (2007)
24. Abujarad, F., Kulkarni, S.S.: Automated constraint-based addition of nonmasking and stabilizing fault-tolerance. Theoretical Computer Science 412(33), 4228–4246 (2011)
25. Farahat, A., Ebnenasir, A.: A lightweight method for automated design of convergence in network protocols. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 7(4), 38:1–38:36 (2012)

26. Ebnenasir, A., Farahat, A.: Swarm synthesis of convergence for symmetric protocols. In: Proceedings of the Ninth European Dependable Computing Conference, pp. 13–24 (2012)
27. Bonakdarpour, B., Kulkarni, S.S.: Revising distributed UNITY programs is NP-complete. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 408–427. Springer, Heidelberg (2008)

Deadlock Checking by Data Race Detection

Ka I Pun[1], Martin Steffen[1], and Volker Stolz[1,2]

[1] University of Oslo, Norway
[2] United Nations University—Intl. Inst. for Software Technology, Macao, China

Abstract. Deadlocks are a common problem in programs with lock-based concurrency and are hard to avoid or even to detect. One way for deadlock prevention is to statically analyze the program code to spot sources of potential deadlocks.

We reduce the problem of deadlock checking to race checking, another prominent concurrency-related error for which good (static) checking tools exist. The transformation uses a type and effect-based static analysis, which analyzes the data flow in connection with lock handling to find out control-points that are potentially part of a deadlock. These control-points are instrumented appropriately with additional shared variables, i.e., race variables injected for the purpose of the race analysis. To avoid overly many false positives for deadlock cycles of length longer than two, the instrumentation is refined by adding "gate locks". The type and effect system and the transformation are formally given. We prove our analysis sound using a simple, concurrent calculus with re-entrant locks.

1 Introduction

Concurrent programs are notoriously hard to get right and at least two factors contribute to this fact: Correctness properties of a parallel program are often global in nature, i.e., result from the correct interplay and cooperation of multiple processes. Hence also violations are non-local, i.e., they cannot typically be attributed to a single line of code. Secondly, the non-deterministic nature of concurrent executions makes concurrency-related errors hard to detect and to reproduce. Since typically the number of different interleavings is astronomical or infinite, testing will in general not exhaustively cover all behavior and errors may remain undetected until the software is in use.

Arguably the two most important and most investigated classes of concurrency errors are *data races* [3] and *deadlocks* [9]. A data race is the simultaneous, unprotected access to mutable shared data with at least one write access. A deadlock occurs when a number of processes are unable to proceed, when waiting cyclically for each other's non-shareable resources without releasing one's own [7]. Deadlocks and races constitute equally pernicious, but complementary hazards: locks offer protection against races by ensuring mutually exclusive access, but may lead to deadlocks, especially using fine-grained locking, or are at least detrimental to the performance of the program by decreasing the degree of parallelism. Despite that, both share some commonalities, too: a race, respectively

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 34–50, 2013.
DOI: 10.1007/978-3-642-40213-5_3,
© IFIP International Federation for Information Processing 2013

a deadlock, manifests itself in the execution of a concurrent program, when two processes (for a race) resp. two or more processes (for a deadlock) reach respective control-points that when reached *simultaneously*, constitute an unfortunate interaction: in case of a race, a read-write or write-write conflict on a shared variable, in case of a deadlock, running jointly into a cyclic wait.

In this paper, we define a static analysis for multi-threaded programs which allows reducing the problem of deadlock checking to race condition checking. The analysis is based on a type and effect system [2] which formalizes the data-flow of lock usages and, in the effects, works with an over-approximation on how often different locks are being held. The information is used to instrument the program with additional variables to signal a race at control points that potentially are involved in a deadlock. Despite the fact that races, in contrast to deadlocks, are a binary global concurrency error in the sense that only two processes are involved, the instrumentation is not restricted to deadlock cycles of length two. To avoid raising too many spurious alarms when dealing with cycles of length > 2, the transformation adds additional locks, to prevent that already parts of a deadlock cycle give raise to a race, thus falsely or prematurely indicating a deadlock by a race.

Our approach widens the applicability of freely available state-of-the-art static race checkers: *Goblint* [20] for the C language, which is not designed to do any deadlock checking, will report appropriate data races from programs instrumented through our transformation, and thus becomes a deadlock checker as well. *Chord* [15] for Java only analyses deadlocks of length two for Java's `synchronized` construct, but not explicit locks from `java.util.concurrent`, yet through our instrumentation reports corresponding races for longer cycles *and* for deadlocks involving explicit locks.

The remainder of the paper is organised as follows. Section 2 presents syntax and operational semantics of the calculus. Afterwards, Section 3 formalizes the data flow analysis in the form of a (constraint-based) effect system. The obtained information is used in Sections 4 and 5 to instrument the program with race variables and additional locks. The sections also prove the soundness of the transformation. We conclude in Section 6 discussing related and future work.

2 Calculus

In this section we present the syntax and (operational) semantics for our calculus, formalizing a simple, concurrent language with dynamic thread creation and higher-order functions. Locks likewise can be created dynamically, they are re-entrant and support non-lexical use of locking and unlocking. The abstract syntax is given in Table 1. A program P consists of a parallel composition of processes $p\langle t\rangle$, where p identifies the process and t is a thread, i.e., the code being executed. The empty program is denoted as $\emptyset$. As usual, we assume $\parallel$ to be associative and commutative, with $\emptyset$ as neutral element. As for the code we distinguish threads t and expressions e, where t basically is a sequential composition of expressions. Values are denoted by v, and $\texttt{let}\ x{:}T = e\ \texttt{in}\ t$ represents

Listing 1. Dining Philosophers

```
let l₁ = new L; ...; lₙ = new L  /* create all locks */
 phil = fun F(x,y) . ( x.lock; y.lock;     /* eat */
                       y.unlock; x.unlock; /* think */
          F(x,y) )
 in spawn(phil(l₁,l₂)); ... ; spawn(phil(lₙ,l₁))
```

Table 1. Abstract syntax

$P ::= \emptyset \mid p\langle t\rangle \mid P \parallel P$	program
$t ::= v \mid \texttt{let}\ x{:}T = e\ \texttt{in}\ t$	thread
$e ::= t \mid v\ v \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{spawn}\ t \mid \texttt{new L} \mid v.\,\texttt{lock} \mid v.\,\texttt{unlock}$	expr.
$v ::= x \mid l^r \mid \texttt{true} \mid \texttt{false} \mid \texttt{fn}\ x{:}T.t \mid \texttt{fun}\ f{:}T.x{:}T.t$	values

the sequential composition of e followed by t, where the eventual result of e, i.e., once evaluated to a value, is bound to the local variable x. Expressions, as said, are given by e, and threads are among possible expressions. Further expressions are function application, conditionals, and the spawning of a new thread, written `spawn` t. The last three expressions deal with lock handling: `new L` creates a new lock (initially free) and gives a reference to it (the `L` may be seen as a class for locks), and furthermore $v.$`lock` and $v.$`unlock` acquires and releases a lock, respectively. Values, i.e., evaluated expressions, are variables, lock references, and function abstractions, where we use `fun` $f{:}T_1.x{:}T_2.t$ for recursive function definitions. Note that the grammar insists that, e.g., in an application, both the function and the arguments are values, analogously when acquiring a lock, etc. This form of representation is known as *a-normal form* [11].

Listing 1 shows the paraphrased code for the well-known Dining Philosopher example. The recursive body used for each philosopher is polymorphic in the lock locations.

The grammar for types, effects, and annotations is given Table 2, where π represents labels (used to label program points where locks are created), r represents (finite) sets of πs, where ρ is a corresponding variable. Labels π are an abstraction of concrete lock references which exist at run-time (namely all those references created at that program point) and therefore we refer to labels π as well as lock sets r also as *abstract locks*. Types include basic types B such as integers, booleans, etc., left unspecified, function types $\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2$, and in particular lock types `L`. To capture the data flow concerning locks, the lock types are annotated with a lock set r, i.e., they are of the form $\texttt{L}^r$. This information will be inferred, and the user, when using types in the program, uses types without annotations (the "underlying" types). We write $T, T_1, T_2, \ldots$ as meta-variables for the underlying types, and $\hat{T}$ and its syntactic variants for the annotated types, as given in the grammar. Furthermore, polymorphism for function definition is captured by type schemes $\hat{S}$, i.e., types prefix-quantified over variables ρ and X, under some constraints. We let Y abbreviate either variables ρ or X, where

Table 2. Types

r	$::=$	$\rho \mid \{\pi\} \mid r \sqcup r$	lock/label sets
$\hat{T}$	$::=$	$B \mid \mathtt{L}^r \mid \hat{T} \xrightarrow{\varphi} \hat{T}$	types
$\hat{S}$	$::=$	$\forall \vec{Y}{:}C.\ \hat{T}$	type schemes
φ	$::=$	$\Delta \to \Delta$	effects/pre- and post specification
Δ	$::=$	$\bullet \mid X \mid \Delta, r{:}n$	lock env./abstract state
C	$::=$	$\emptyset \mid \rho \sqsupseteq r,\ C \mid X \geq \Delta, C$	constraints

X is a variable for effect which is introduced later. Any specialization of the type scheme $\forall \vec{Y}{:}C.\hat{T}$ has to satisfy the constraints C. For the deadlock and race analysis we need not only information which locks are used where, but also an estimation about the "value" of the lock, i.e., how often the abstractly represented locks are taken.

Estimation of the lock values, resp. their change is captured in the behavioral *effects* φ in the form of pre- and post-specifications $\Delta_1 \to \Delta_2$. Abstract states (or lock environments) Δ are of the form $r_0{:}n_0, r_1{:}n_1, \ldots$. The constraint based type system works on lock environments using variables only, i.e., the Δ are of the form $\rho_0{:}n_0, \rho_1{:}n_1, \ldots$, maintaining that each variable occurs at most once. Thus, in the type system, the environments Δ are mappings from variables ρ to lock counter values n, where n ranges from $+\infty$ to $-\infty$. As for the syntactic representation of those mappings: we assume that a variable ρ *not* mentioned in Δ corresponds to the binding $\rho{:}0$, e.g. in the empty mapping $\bullet$. Constraints C finally are finite sets of subset inclusions of the forms $\rho \sqsupseteq r$ and $X \geq \Delta$. We assume that the user provides the underlying types, i.e., without location and effect annotation, while our type system in Section 3 derives the smallest possible type in terms of originating locations for each variable of lock-type `L` in the program.

Semantics

Next we present the operational semantics, given in the form of a small-step semantics, distinguishing between local and global steps (cf. Tables 3 and 4). The local semantics deals with reduction steps of one single thread of the form $t_1 \to t_2$. Rule R-Red is the basic evaluation step which replaces the local variable in the continuation thread t by the value v (where $[v/x]$ represents capture-avoiding substitution). The Let-construct generalizes sequential composition and rule R-Let restructures a nested let-construct expressing associativity of that construct. Thus it corresponds to transforming $(e_1; t_1); t_2$ into $e_1; (t_1; t_2)$. Together with the first rule, it assures a deterministic left-to-right evaluation within each thread. The two R-If-rules cover the two branches of the conditional and the R-App-rules deals with function application (of non-recursive, resp. recursive functions).

The global steps are given in Table 4, formalizing transitions of configurations of the form $\sigma \vdash P$, i.e., the steps are of the form $\sigma \vdash P \to \sigma' \vdash P'$, where P is a

Table 3. Local steps

$\texttt{let}\ x{:}T = v\ \texttt{in}\ t \ \rightarrow\ t[v/x]$ R-RED

$\texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2 \ \rightarrow \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2 = t_1\ \texttt{in}\ t_2)$ R-LET

$\texttt{let}\ x{:}T = \texttt{if true then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t \ \rightarrow \texttt{let}\ x{:}T = e_1\ \texttt{in}\ t$ R-IF$_1$

$\texttt{let}\ x{:}T = \texttt{if false then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t \ \rightarrow \texttt{let}\ x{:}T = e_2\ \texttt{in}\ t$ R-IF$_2$

$\texttt{let}\ x{:}T = (\texttt{fn}\ x'{:}T'.t')\ v\ \texttt{in}\ t \rightarrow \texttt{let}\ x{:}T = t'[v/x']\ \texttt{in}\ t$ R-APP$_1$

$\texttt{let}\ x{:}T = (\texttt{fun}\ f{:}T_1.x'{:}T_2.t')\ v\ \texttt{in}\ t \ \rightarrow \texttt{let}\ x{:}T = t'[v/x'][\texttt{fun}\ f{:}T_1.x'{:}T_2.t'/f]\ \texttt{in}\ t$ R-APP$_2$

Table 4. Global steps

$$\frac{t_1 \rightarrow t_2}{\sigma \vdash p\langle t_1\rangle \rightarrow \sigma \vdash p\langle t_2\rangle}\ \text{R-LIFT} \qquad \frac{\sigma \vdash P_1 \rightarrow \sigma' \vdash P_1'}{\sigma \vdash P_1 \parallel P_2 \rightarrow \sigma' \vdash P_1' \parallel P_2}\ \text{R-PAR}$$

$$\sigma \vdash p_1\langle \texttt{let}\ x{:}T = \texttt{spawn}\ t_2\ \texttt{in}\ t_1\rangle \rightarrow \sigma \vdash p_1\langle \texttt{let}\ x{:}T = p_2\ \texttt{in}\ t_1\rangle \parallel p_2\langle t_2\rangle \quad \text{R-SPAWN}$$

$$\frac{\sigma' = \sigma[l \mapsto \mathit{free}] \qquad l \text{ is fresh}}{\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{new L in}\ t\rangle \rightarrow \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-NEWL}$$

$$\frac{\sigma(l) = \mathit{free} \vee \sigma(l) = p(n) \qquad \sigma' = \sigma +_p l}{\sigma \vdash p\langle \texttt{let}\ x{:}T = l.\ \texttt{lock in}\ t\rangle \rightarrow \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-LOCK}$$

$$\frac{\sigma(l) = p(n) \qquad \sigma' = \sigma -_p l}{\sigma \vdash p\langle \texttt{let}\ x{:}T = l.\ \texttt{unlock in}\ t\rangle \rightarrow \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-UNLOCK}$$

program, i.e., the parallel composition of a finite number of threads running in parallel, and σ is a finite mapping from lock identifiers to the status of each lock (which can be either free or taken by a thread where a natural number indicates how often a thread has acquired the lock, modelling re-entrance). Thread-local steps are lifted to the global level by R-LIFT. Rule R-PAR specifies that the steps of a program consist of the steps of the individual threads, sharing σ. Executing the spawn-expression creates a new thread with a fresh identity which runs in parallel with the parent thread (cf. rule R-SPAWN). Globally, the process identifiers are unique. A new lock is created by `newL` (cf. rule R-NEWL) which allocates a fresh lock reference in the heap. Initially, the lock is free. A lock l is acquired by executing $l.$`lock`. There are two situations where that command does not block, namely the lock is free or it is already held by the requesting process p. The heap update $\sigma +_p l$ is defined as follows: If $\sigma(l) = \mathit{free}$, then $\sigma +_p l = \sigma[l \mapsto p(1)]$ and if $\sigma(l) = p(n)$, then $\sigma +_p l = \sigma[l \mapsto p(n+1)]$. Dually $\sigma -_p l$ is defined as follows: if $\sigma(l) = p(n+1)$, then $\sigma -_p l = \sigma[l \mapsto p(n)]$, and if $\sigma(l) = p(1)$, then $\sigma -_p l = \sigma[l \mapsto \mathit{free}]$. Unlocking works correspondingly, i.e., it sets the lock as being free resp. decreases the lock count by one (cf. rule R-UNLOCK). In the premise of the rules it is checked that the thread performing the unlocking actually holds the lock.

To analyze deadlocks and races, we specify which locks are meant statically by labelling the program points of lock creations with π, i.e., lock creation statements `new L` are augmented to $\texttt{new}_\pi$ `L` where the annotations π are assumed unique for a given program. We assume further that the lock references l are also labelled l^ρ; the labelling is done by the type system presented next.

3 Type and Effect System

Next we present a constraint-based type and effect system for information which locks are being held at various points in the code. The analysis works thread-locally, i.e., it analyzes the code of one thread. In Section 4, we will use this information to determine points in a program, that globally may lead to deadlocks and which are then instrumented appropriately by additional race variables. The judgments of the system are of the form

$$\Gamma \vdash e : \hat{T} :: \rho; C \ , \tag{1}$$

where ρ is of the form $\Delta_1 \rightarrow \Delta_2$. Equivalently, we write also $\Gamma; \Delta_1 \vdash e : \hat{T} :: \Delta_2; C$ for the judgment. The judgment expresses that e is of type $\hat{T}$, where for annotated lock types of the form $\texttt{L}^r$ the r expresses the potential points of creation of the lock. The effect $\varphi = \Delta_1 \rightarrow \Delta_2$ expresses the change in the lock counters, where Δ_1 is the pre-condition and Δ_2 the post-condition (in a partial correctness manner). The types and the effects contain variables ρ and X; hence the judgement is interpreted relative to the solutions of the set of constraints C. Given Γ and e, the constraint set C is generated during the derivation. Furthermore, the pre-condition Δ_1 is considered as given, whereas Δ_2 is derived.

The rules for the type system are given in Table 5. The rule TA-Var combines looking up the variable from the context with instantiation, choosing fresh variables to assure that the constraints θC, where C is taken from the variable's type scheme, are most general. As a general observation and as usual, values have no effect, i.e., its pre- and post-condition are identical. Also lock creation in rule TA-NewL does not have an effect. As for the flow: π labels the point of creation of the lock; hence a new constraint is generated, requiring $\rho \sqsupseteq \{\pi\}$ for the ρ-annotation in the lock type. The case for lock references l^ρ in rule TA-LRef works analogously, where the generated constraint uses the lock variable ρ instead of the concrete point of creation.

For function abstraction in rule TA-Abs$_1$, the premise checks the body e of the function with the typing context extended by $x{:}\lceil T \rceil_A$, where the operation $\lceil T \rceil_A$ turns all occurrences of lock types `L` in T into their annotated counterparts using *fresh* variables, as well as introducing state variables for the latent effects of higher-order functions. Also for the pre-condition of the function body, a fresh variable is used. The recursive function is also formulated similarly. It uses in addition a *fresh* variable for the post-condition of the function body, and constraints requiring $X_2 \geq \Delta_2$ and $\hat{T}_2 \geq \hat{T}_2'$ are generated. For function application (cf. rule TA-App), the subtyping requirement between the type $\hat{T}_2$

Table 5. Constraint based type and effect system

$$\frac{\Gamma(x) = \forall \vec{Y}{:}C.\hat{T} \qquad \vec{Y}' \textit{ fresh} \qquad \theta = [\vec{Y}'/\vec{Y}]}{\Gamma \vdash x : \theta\hat{T} :: \Delta \to \Delta; \theta C}\ \textsc{TA-Var} \qquad \frac{\Gamma \vdash t : \hat{T} :: \bullet \to \Delta_2; C}{\Gamma \vdash \texttt{spawn}\, t : \texttt{Thread} :: \Delta_1 \to \Delta_1; C}\ \textsc{TA-Spawn}$$

$$\frac{\rho \textit{ fresh}}{\Gamma \vdash \texttt{new}_\pi\ \texttt{L} : \texttt{L}^\rho :: \Delta \to \Delta; \rho \sqsupseteq \{\pi\}}\ \textsc{TA-NewL} \qquad \frac{\rho' \textit{ fresh}}{\Gamma \vdash l^\rho : \texttt{L}^{\rho'} :: \Delta \to \Delta; \rho' \sqsupseteq \rho}\ \textsc{TA-LRef}$$

$$\frac{\hat{T}_1 = \lceil T_1 \rceil_A \qquad \Gamma, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: X \to \Delta_2; C \qquad X \textit{ fresh}}{\Gamma \vdash \texttt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{X \to \Delta_2} \hat{T}_2 :: \Delta_1 \to \Delta_1; C}\ \textsc{TA-Abs}_1$$

$$\frac{\begin{array}{c}\hat{T}_1 = \lceil T_1 \rceil_A \qquad \hat{T}_2 = \lceil T_2 \rceil_A \qquad X_1, X_2 \textit{ fresh} \\ \Gamma, f{:}\hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x{:}\hat{T}_1 \vdash e : \hat{T}_2' :: X_1 \to \Delta_2; C_1 \qquad \hat{T}_2' \leq \hat{T}_2 \vdash C_2 \qquad C_3 = \Delta_2 \leq X_2\end{array}}{\Gamma \vdash \texttt{fun}\, f{:}T_1 \to T_2, x{:}T_1.e : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1; C_1, C_2, C_3}\ \textsc{TA-Abs}_2$$

$$\frac{\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta \to \Delta; C_1 \qquad \Gamma \vdash v_2 : \hat{T}_2' :: \Delta \to \Delta; C_2 \qquad \hat{T}_2' \leq \hat{T}_2 \vdash C \qquad X \textit{ fresh}}{\Gamma \vdash v_1\ v_2 : \hat{T}_1 :: \Delta \to X; C_1, C_2, C, \Delta \leq \Delta_1, \Delta_2 \leq X}\ \textsc{TA-App}$$

$$\frac{\begin{array}{c}T = \lfloor \hat{T}_1 \rfloor = \lfloor \hat{T}_2 \rfloor \qquad \hat{T}; C = \hat{T}_1 \vee \hat{T}_2 \qquad \Delta'; C' = \Delta_1' \vee \Delta_2' \\ \Gamma \vdash v : \texttt{Bool} :: \Delta_0 \to \Delta_0; C_0 \quad \Gamma \vdash e_1 : \hat{T}_1 :: \Delta_0 \to \Delta_1; C_1 \quad \Gamma \vdash e_2 : \hat{T}_2 :: \Delta_0 \to \Delta_2; C_2\end{array}}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \Delta_0 \to \Delta'; C_0, C_1, C_2, C, C'}\ \textsc{TA-Cond}$$

$$\frac{\Gamma \vdash e_1 : \hat{T}_1 :: \Delta_1 \to \Delta_2; C_1 \qquad \lfloor \hat{T}_1 \rfloor = T_1 \qquad \hat{S}_1 = \textit{close}(\Gamma, C_1, \hat{T}_1) \qquad \Gamma, x{:}\hat{S}_1 \vdash e_2 : \hat{T}_2 :: \Delta_2 \to \Delta_3; C_2}{\Gamma \vdash \texttt{let}\, x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \Delta_1 \to \Delta_3; C_2}\ \textsc{TA-Let}$$

$$\frac{\Gamma \vdash v : \texttt{L}^\rho :: \Delta \to \Delta; C_1 \qquad X \textit{ fresh} \qquad C_2 = X \geq \Delta \oplus (\rho{:}1)}{\Gamma \vdash v.\,\texttt{lock} : \texttt{L}^\rho :: \Delta \to X; C_1, C_2}\ \textsc{TA-Lock}$$

$$\frac{\Gamma \vdash v : \texttt{L}^\rho :: \Delta \to \Delta; C_1 \qquad X \textit{ fresh} \qquad C_2 = X \geq \Delta \ominus (\rho{:}1)}{\Gamma \vdash v.\,\texttt{unlock} : \texttt{L}^\rho :: \Delta \to X; C_1, C_2}\ \textsc{TA-Unlock}$$

of the argument and the function's input type $\hat{T}_2'$ is used to generate additional constraints. Furthermore, the precondition Δ of the application is connected with the precondition of the latent effect Δ_1 and the post-condition of the latent effect with the post-condition of the application, the latter one using again a fresh variable. The corresponding two constraints $\Delta \leq \Delta_1$ and $\Delta_2 \leq X$ represent the control flow when calling, resp. when returning to the call site. The treatment of conditionals is standard (cf. rule TA-Cond). To assure that the resulting type is an upper bound for the types of the two branches, two additional constraints C and C' are generated.

The let-construct (cf. rule TA-Let) is combined with the rule for generalization, such that for checking the body e_2, the typing context is extended by a type scheme $\hat{S}_1$ which generalizes the type $\hat{T}_1$ of expression e_1. The close-operation is defined as $\textit{close}(\Gamma, C, \hat{T}) = \forall \vec{Y}{:}C.\hat{T}$ where the quantifier binds all variables occurring free in C and $\hat{T}$ but not in Γ. Spawning a thread in rule TA-Spawn has no effect, where the premise of the rule checks well-typedness of the thread being spawned. The last two rules deal with locking and unlocking,

simply counting up, resp. down the lock counter, setting the post-condition to over-approximate $\Delta \oplus \rho$, resp. $\Delta \ominus \rho$.

The type system is basically a single-threaded analysis. For subject reduction later and soundness of the analysis, we also need to analyse processes running in parallel. The definition is straightforward, since a global program is well-typed simply if all its threads are. For one thread, $p\langle t\rangle : p\langle\varphi_k; C\rangle$, if $\vdash t : \hat{T} :: \varphi; C$ for some type $\hat{T}$. We will abbreviate $p_1\langle\varphi_1; C_1\rangle \parallel \ldots \parallel p_k\langle\varphi_k; C_k\rangle$ by Φ.

Constraints C come in two forms: $r \sqsubseteq \rho$ and $X_1 \leq X_2 \oplus (\rho{:}n)$ resp. $X_1 \leq X_2 \ominus (\rho{:}n)$. We consider both kinds of constraints as independent, in particular a constraint of the form $X_1 \leq X_2 \oplus (\rho{:}n)$ is considered as a constraint between the two variables X_1 and X_2 and not as a constraint between X_1, X_2, *and* ρ. Given C, we write C^ρ for the ρ-constraints in C and C^X for the constraints concerning X-variables. Solutions to the constraints are ground substitutions; we use θ to denote substitutions. analogous to the distinction for the constraints, we write θ^ρ for substitutions concerning the ρ-variables and θ^X for substitutions concerning the X-variables. A ground θ^ρ-substitution maps ρ's to finite sets $\{\pi_1, \ldots, \pi_n\}$ of labels and a ground θ^X-substitution maps X's to Δ's (which are of the form $\rho_1{:}n_1, \ldots, \rho_k{:}n_k$); note that the range of the ground θ^X-substitution still contains ρ-variables. We write $\theta^\rho \models C$ if θ^ρ solves C^ρ and analogously $\theta^X \models C$ if θ^X solves C^X. For a $\theta = \theta^X\theta^\rho$, we write $\theta \models C$ if $\theta^\rho \models C$ and $\theta^X \models C$. Furthermore we write $C_1 \models C_2$ if $\theta \models C_1$ implies $\theta \models C_2$, for all ground substitutions θ. For the simple super-set constraints of the form $\rho \sqsupseteq r$, constraints always have a unique minimal solution. Analogously for the C^X-constraints. A heap σ satisfies an abstract state Δ, if Δ over-approximates the lock counter for all locks in σ: Assuming that Δ does not contain any ρ-variables and that the lock references in σ are labelled by π's, $\sigma \models \Delta$ if $\sum_{\pi\in r} \sigma(l^\pi) \leq \Delta(r)$ (for all r in $dom(\Delta)$). Given a constraint set C, an abstract state Δ (with lock references l^ρ labelled by variables) and a heap σ, we write $\sigma \models_C \Delta$ ("σ satisfies Δ under the constraints C"), iff $\theta \models C$ implies $\theta\sigma \models \theta\Delta$, for all θ. A heap σ satisfies a global effect Φ (written $\sigma \models \Phi$), if $\sigma \models_{C_i} \Delta_i$ for all $i \leq k$ where $\Phi = p_1\langle\varphi_1; C_1\rangle \parallel \ldots \parallel p_k\langle\varphi_k; C_k\rangle$ and $\varphi_i = \Delta_i \to \Delta'_i$.

Soundness

Next we prove soundness of the analysis wrt. the semantics. The core of the proof is the preservation of well-typedness under reduction ("subject reduction"). The static analysis does not only give back types (as an abstraction of resulting *values*) but also effects (in the form of pre- and post-specification). While types are preserved, we cannot expect that the effect of an expression remains unchanged under reduction. As the pre- and post-conditions specify (upper bounds on) the allowed lock values, the only steps which change are locking and unlocking steps. To relate the change of pre-condition with the steps of the system we assume the transitions to be labelled. Relevant is only the lock set variable ρ; the identity p of the thread, the label π and the actual identity of the lock are not relevant for the formulation of subject reduction, hence we do not include that information in the labels here. The steps for lock-taking are of the form $\sigma_1 \vdash p\langle t_1\rangle \xrightarrow{p\langle\rho.\mathtt{lock}\rangle} \sigma_2 \vdash p\langle t_2\rangle$;

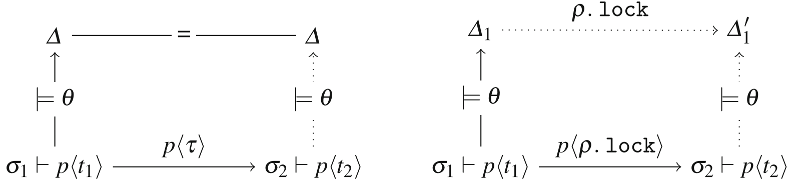


Fig. 1. Subject reduction (case of unlocking analogous)

unlocking steps analogously are labelled by $\rho.\,\texttt{unlock}$ and all other steps are labelled by τ, denoting internal steps. The formulation of subject reduction can be seen as a form of *simulation* (cf. Figure 1): The concrete steps of the system —for one process in the formulation of subject reduction— are (weakly) simulated by changes on the abstract level; weakly, because τ-steps are ignored in the simulation. To make the parallel between simulation and subject reduction more visible, we write $\Delta_1 \xrightarrow{\rho.\texttt{lock}} \Delta_2$ for $\Delta_2 = \Delta_1 \oplus \rho$ (and analogously for unlocking).

Lemma 1. (Subject reduction) *Assume* $\Gamma \vdash P \parallel p\langle t_1\rangle :: \Phi \parallel p\langle \Delta_1 \to \Delta_2; C_1\rangle$*, and furthermore* $\theta \models C_1$ *for some ground substitution and* $\sigma_1 \models \theta\Delta_1$ *and* $\sigma_1 \models \Phi$*.*

1. $\sigma_1 \vdash P \parallel p\langle t_1\rangle \xrightarrow{p\langle\tau\rangle} \sigma_2 \vdash P \parallel p\langle t_2\rangle$*, then* $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1' \to \Delta_2', C_2\rangle$ *where* $C_1 \vdash \Delta_1 \leq \Delta_1'$ *and* $C_1 \vdash \Delta_2' \leq \Delta_2$*. Furthermore,* $C_1 \models C_2$ *and* $\sigma_2 \models \theta\Delta_1$ *and* $\sigma_2 \models \Phi$*.*
2. $\sigma_1 \vdash P \parallel p\langle t_1\rangle \xrightarrow{p\langle\rho.\texttt{lock}\rangle} \sigma_2 \vdash P \parallel p\langle t_2\rangle$*, then* $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1' \to \Delta_2, C_2\rangle$ *where* $C_1 \vdash \Delta_1 \oplus \rho \leq \Delta_1'$ *and* $C_1 \vdash \Delta_2' \leq \Delta_2$*. Furthermore* $C_1 \models C_2$ *and* $\sigma_2 \models \theta\Delta_1'$ *and* $\sigma_2 \models \Phi$*.*
3. $\sigma_1 \vdash P \parallel p\langle t_1\rangle \xrightarrow{p\langle\rho.\texttt{unlock}\rangle} \sigma_2 \vdash P \parallel p\langle t_2\rangle$*, then* $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1' \to \Delta_2, C_2\rangle$ *where* $C_1 \vdash \Delta_1 \ominus \rho \leq \Delta_1'$ *and* $C_1 \vdash \Delta_2' \leq \Delta_2$*. Furthermore* $C_1 \models C_2$ *and* $\sigma_2 \models \theta\Delta_1'$ *and* $\sigma_2 \models \Phi$*.*

The property of the lemma is shown pictorially in Figure 1.

As an immediate consequence, all configurations reachable from a well-typed initial configuration are well-typed itself. In particular, for all those reachable configurations, the corresponding pre-condition (together with the constraints) is a sound over-approximation of the actual lock counters in the heap.

Corollary 1. (Soundness of the approximation) *Let* $\sigma_0 \vdash p\langle t_0\rangle$ *be an initial configuration. Assume further* $\Gamma \vdash p\langle t_0\rangle :: p\langle \Delta_0 \to \Delta_2; C\rangle$ *and* $\theta \models C$ *and where* Δ_0 *is the empty context. If* $\sigma_0 \vdash p\langle t_0\rangle \to^* \sigma \vdash P$*, then* $\Gamma \vdash P :: \Phi$*, where* $\Phi = p_1\langle \Delta_1 \to \Delta_1'; C_1\rangle \parallel \ldots \parallel p_k\langle \Delta_k \to \Delta_k'; C_k\rangle$ *and where* $\sigma \models \theta\Delta_i$ *(for all i).*

4 Race Variables for Deadlock Detection

Next we use the information inferred by the type system in the previous section to locate control points in a program which potentially give rise to a deadlock.

As we transform the given program after analyzing it, for improved precision, we assume that in the following all non-recursive function applications are instantiated/ inlined: a unique call-site per function ensures the most precise type- and effect information for that function, and correspondingly the best suitable instrumentation. The polymorphic type system gives a context-sensitive representation, which can then be instantiated per call-site. Note that this way, we need to analyze only the original program, and each function in there once, although for the next step, we duplicate methods. Recursive functions are instantiated once with (minimal) effects capturing all call-sites.

Those points are instrumented appropriately with assignments to additional shared variables, intended to flag a race. To be able to do so, we slightly need to extend our calculus. The current formulation does not have shared variables, as they are irrelevant for the analysis of the program, which concentrates on the locks. In the following we assume that we have appropriate syntax for accessing shared variables; we use $z, z', z_1, \ldots$ to denote shared variables, to distinguish them from the let-bound thread-local variables x and their syntactic variants. For simplicity, we assume that they are statically and globally given, i.e., we do not introduce syntax to declare them. Together with the lock references, their values are stored in σ. To reason about changes to those shared variables, we introduce steps of the form $\xrightarrow{p\langle !z\rangle}$ and $\xrightarrow{p\langle ?z\rangle}$, representing write resp. read access of process p to z. Alternatives to using a statically given set of shared variables, for instance using dynamically created pointers to the heaps are equally straightforward to introduce syntactically and semantically, without changing the overall story.

4.1 Deadlocks and Races

We start by formally defining the notion of deadlock used here, which is fairly standard (see also [16]): a program is deadlocked, if a number of processes are cyclically waiting for each other's locks.

Definition 1. (Waiting for a lock) *Given a configuration $\sigma \vdash P$, a process p* waits *for a lock l in $\sigma \vdash P$, written as waits$(\sigma \vdash P, p, l)$, if (1) it is not the case that $\sigma \vdash P \xrightarrow{p\langle l.\mathtt{lock}\rangle}$, and furthermore (2) there exists σ' s.t. $\sigma' \vdash P \xrightarrow{p\langle l.\mathtt{lock}\rangle} \sigma'' \vdash P'$. In a situation without (1), we say that in configuration $\sigma \vdash P$, process p* tries *for lock l (written tries$(\sigma \vdash P, p, l)$).*

Definition 2. (Deadlock) *A configuration $\sigma \vdash P$* is deadlocked *if $\sigma(l_i) = p_i(n_i)$ and furthermore waits$(\sigma \vdash P, p_i, l_{i+_k 1})$ (where $k \geq 2$ and for all $0 \leq i \leq k-1$). The $+_k$ is meant as addition modulo k. A configuration $\sigma \vdash P$* contains a deadlock*, if, starting from $\sigma \vdash P$, a deadlocked configuration is reachable; otherwise it is deadlock free.*

Thus, a process can only be deadlocked, i.e., being part of a deadlocked configuration, if p *holds* at least one lock already, and is *waiting* for another one. With re-entrant locks, these two locks must be different. Independent from whether it leads to a deadlock or not, we call such a situation —holding a lock

and attempting to acquire another one— a *second lock point.* More concretely, given a configuration, where we abbreviate the situation where process p holds lock l_1 and *tries* l_2 by $slp(\sigma \vdash P)_p^{l_1 \to l_2}$. The abstraction in the analysis uses program points π to represent concrete locks, and the goal thus is to detect in an approximate manner cycles using those abstractions π. As stated, a concrete deadlock involves a cycle of processes and locks. We call an *abstract cycle* Δ_C a sequence of pairs $\vec{p}{:}\vec{\pi}$ with the interpretation that p_i is holding π_i and wants π_{i+1} (modulo the length of the cycle). Next we fix the definition for being a second lock point. At run-time a process is at a second lock point simply if it holds a lock and tries to acquire a another, different one.

Definition 3. (Second lock point (runtime)) *A local configuration $\sigma \vdash p\langle t\rangle$ is at a second point (holding l_1 and attempting l_2, when specific), written $slp(\sigma \vdash p\langle t\rangle)^{l_1 \to l_2}$, if $\sigma(l_1) = p(n)$ and $tries(\sigma \vdash p\langle t\rangle, l_2)$. Analogously for abstract locks and heaps over those: $slp(\sigma \vdash p\langle t\rangle)^{\pi_1 \to \pi_2}$, if $\sigma(\pi_1) = p(n)$ and $tries(\sigma \vdash p\langle t\rangle, \pi_2)$. Given an abstract cycle Δ_C a local configuration is at a second lock point of Δ_C, if $slp(\sigma \vdash p\langle t\rangle)^{\pi_1 \to \pi_2}$ where, as specified by Δ_C, p holds π_1 and wants π_2. Analogously we write for global configurations e.g., $slp(\sigma \vdash P)_p^{\pi_1 \to \pi_2}$, where p is the identity of a thread in P.*

Ultimately, the purpose of the static analysis is to derive (an over-approximation of the) second lock points as a basis to instrument with race variables. The type system works thread-locally, i.e., it derives potential second lock points *per thread.* Given a static thread, i.e., an expression t without run-time syntax, second lock points are control points where the static analysis derives the danger of attempting a second lock. A control-point in a thread t corresponds to the *occurrence* of a sub-expression; we write $t[t']$ to denote the occurrence of t' in t. As usual, occurrences are assumed to be unique.

Definition 4. (Second lock point (static)) *Given a static thread $t_0[t]$, a process identifier p and $\Delta_0 \vdash_p t_0 : \Delta$, where $\Delta_0 = \bullet$. The occurrence of t in t_0 is a* static slp *if:*

1. *$t = \texttt{let}\ x{:}\texttt{L}^{\{\dots,\pi,\dots\}} = v.\ \texttt{lock in}\ t'$.*
2. *$\Delta_1 \vdash_p t :: \Delta_2$, for some Δ_1 and Δ_2, occurs in a sub-derivation of $\Delta_0 \vdash t_0 :: \Delta$.*
3. *there exists $\pi' \in \Delta_1$ s.t. $\Delta_C \vdash p$ has π', and $\Delta_C \vdash p$ wants π .*

Assume further $\sigma_0 \vdash p\langle t_0\rangle \longrightarrow^* \sigma \vdash p\langle t\rangle$. We say $\sigma \vdash p\langle t\rangle$ is at a static second lock point if t occurs as static second lock point in t_0.

Lemma 2. (Static overapproximation of slp's) *Given Δ_C and $\sigma \vdash P$ be a reachable configuration where $P = P' \parallel p\langle t\rangle$ and where furthermore the initial state of p is $p\langle t_0\rangle$. If $\sigma \vdash p\langle t\rangle$ is at a dynamic slp (wrt. Δ_C), then t is a static slp (wrt. Δ_C).*

Proof. A direct consequence of soundness of the type system (cf. Corollary 1). □

Next we define the notion of *race.* A race manifests itself, if at least two processes in a configuration attempt to access a shared variables at the same time, where at least one access is a write-access.

Definition 5. (Race) *A configuration* $\sigma \vdash P$ *has a (manifest)* race, *if* $\sigma \vdash P \xrightarrow{p_1\langle !x\rangle}$, *and* $\sigma \vdash P \xrightarrow{p_2\langle !x\rangle}$ *or* $\sigma \vdash P \xrightarrow{p_2\langle ?x\rangle}$, *for two different* p_1 *and* p_2. *A configuration* $\sigma \vdash P$ *has a race if a configuration is* reachable *where a race manifests itself. A program has a race, if its initial configuration has a race; it is race-free else.*

Race variables will be added to a program to assure that, if there is a deadlock, also a race occurs. More concretely, being based on the result of the static analysis, appropriate race variables are introduced for each *static* second lock points, namely immediately preceding them. Since static lock points over-approximate the dynamic ones and since being at a dynamic slp is a necessary condition for being involved in a deadlock, that assures that no deadlock remains undetected when checking for races. In that way, that the additional variables "protect" the second lock points.

Definition 6. (Protection) *A property* ψ *is protected by a variable* z *starting from configuration* $\sigma \vdash p\langle t\rangle$, *if* $\sigma \vdash p\langle t\rangle \rightarrow^* \xrightarrow{a} \sigma' \vdash p\langle t'\rangle$ *and* $\psi(p\langle t'\rangle)$ *implies that* $a = !z$. *We say,* ψ *is protected by* z, *if it is protected by* z *starting from an arbitrary configuration.*

Protection, as just defined, refers to a property and the execution of a single thread. For race checking, it must be assured that the local properties are protected by the same, i.e., shared variable are necessarily and commonly reached. That this is the case is formulated in the following lemma:

Lemma 3. (Lifting) *Assume two processes* $p_1\langle t_1\rangle$ *and* $p_2\langle t_2\rangle$ *and two thread-local properties* ψ_1 *and* ψ_2 *(for* p_1 *and* p_2*, respectively). If* ψ_1 *is protected by* x *for* $p_1\langle t_1\rangle$ *and* ψ_2 *for* $p_2\langle t_2\rangle$ *by the same variable, and a configuration* $\sigma \vdash P$ *with* $P = p_1\langle t_1\rangle \parallel p_2\langle t_2\rangle \parallel P''$ *is reachable from* $\sigma' \vdash P'$ *such that* $\psi_1 \wedge \psi_2$ *holds, then* $\sigma' \vdash P'$ *has a race.*

4.2 Instrumentation

Next we specify how to transform the program by adding race variables. The idea is simple: each static second lock point, as determined statically by the type system, is instrumented by an appropriate race variable, adding it in front of the second lock point. In general, to try to detect different potential deadlocks at the same time, different race variables may be added simultaneously (at different points in the program). The following definition defines where to add a race variable representing one particular cycle of locks Δ_C. Since the instrumentation is determined by the static type system, one may combine the derivation of the corresponding lock information by the rules of Table 5 such that the result of the derivation not only derives type and effect information, but also transforms the

program at the same time, with judgments of the form $\Gamma \vdash t \rhd t' : \hat{T} :: \varphi$, where t is transformed to t'. Note that we assume that a solution to the *constraint set has been determined and applied* to the type and the effects. Since the only control points in need of instrumentation are where a lock is taken, the transformation for all syntactic constructs is trivial, leaving the expression unchanged, except for $v.$`lock`-expressions, where the additional assignment is added if the condition for static slp is satisfied (cf. Definition 4).

Definition 7. (Transformation) *Given an abstract cycle Δ_C. For a process p from that cycle, the control points instrumented by a $!z$ are defined as follows:*

$$\dfrac{\dfrac{\Gamma \vdash v : \mathtt{L}^r :: \Delta_1 \to \Delta_1 \quad \Delta_2 = \Delta_1 \oplus r \quad \pi \in r \quad \pi' \in \Delta_1 \quad \Delta_C \vdash p \ \mathit{wants}\ \pi \quad \Delta_C \vdash p\ \mathit{has}\ \pi'}{\Gamma \vdash v.\,\mathtt{lock} : \mathtt{L}^r :: \Delta_1 \to \Delta_2} \qquad \Gamma, x{:}\mathtt{L}^r \vdash t \rhd t' : T :: \Delta_2 \to \Delta_3}{\Gamma \vdash \mathtt{let}\ x{:}T = v.\,\mathtt{lock}\ \mathtt{in}\ t \ \rhd \ \mathtt{let}\ x{:}T = (!z; v.\,\mathtt{lock})\ \mathtt{in}\ t' : T :: \Delta_1 \to \Delta_3}$$

By construction, the added race variable protects the corresponding static slp, and thus, ultimately the corresponding dynamic slp's, as the static ones over-approximate the dynamic ones.

Lemma 4. (Race variables protect slp's) *Given a cycle Δ_C and a corresponding transformed program. Then all static second lock points in the program are protected by the race variable (starting from the initial configuration).*

The next lemma shows that there is a race "right in front of" a deadlocked configuration for a transformed program.

Lemma 5. *Given an abstract cycle Δ_C, and let P_0 be a transformed program according to Definition 7. If the initial configuration $\sigma_0 \vdash P_0$ has a deadlock wrt. Δ_C, then $\sigma_0 \vdash P_0$ has a race.*

Proof. By the definition of deadlock (cf. Definition 2), some deadlocked configuration $\sigma' \vdash P'$ is reachable from the initial configuration:

$$\sigma_0 \vdash P_0 \longrightarrow^* \sigma' \vdash P' \quad \text{where} \quad P' = \ldots p_i\langle t'_i\rangle \parallel \ldots \parallel p_j\langle t'_j\rangle \parallel \ldots\ , \qquad (2)$$

where by assumption, the processes p_i and the locks they are holding, resp. on which they are blocked are given by Δ_C, i.e., $\sigma(l_i) = p_i(n_i)$ and $\mathit{waits}(\sigma' \vdash P', p_i, l_{i+_k1})$. Clearly, each participating process $\sigma' \vdash p_i\langle t'_i\rangle$ is at a *dynamic* slp (cf. Definition 3). Since those are over-approximated by their static analogues (cf. Lemma 2), the occurrence of t'_i in t^0_i resp. of t'_j in t^0_j is a *static* slp. By Lemma 4, all static slp (wrt. the given cycle) are protected, starting from the initial configuration, by the corresponding race variable. This together with the fact that $\sigma' \vdash p_i\langle t'_i\rangle$ is reachable from $\sigma_0 \vdash p_i\langle t^0_i\rangle$ implies that the static slp in each process p_i is protected by the same variable x. Hence, by Lemma 3, $\sigma_0 \vdash P_0$ has a race between p_i and p_j. □

The previous lemma showed that the race variables are added at the "right places" to detect deadlocks. Note, however, that the property of the lemma was formulated for the transformed program while, of course, we intend to detect deadlocks in the original program. So to use the result of Lemma 5 on the original program, we need to convince ourselves that the transformation does not change (in a relevant way) the behavior of the program, in particular that it neither introduces nor removes deadlocks. Since the instrumentation only adds variables which do not influence the behavior, this preservation behavior is obvious. The following lemma shows that transforming programs by instrumenting race variables preserves behavior.

Lemma 6. (Transformation preserves behavior) *P is deadlock-free iff P^T is deadlock-free, for arbitrary programs.*

Next, we state that with the absence of data race in a transformed program, the corresponding original one is deadlock-free:

Lemma 7. (Data races and deadlocks) *P is deadlock-free if P^T is race-free, for arbitrary programs.*

5 Gate Locks

Next we refine the transformation to improve its precision. By definition, races are inherently *binary*, whereas deadlocks in general are not, i.e., there may be more than two processes participating in a cyclic wait. In a transformed program, all the processes involved in a specific abstract cycle Δ_C share a common race variable. While sound, this would lead to unnecessarily many false alarms, because already if two processes as part of a cycle of length $n > 2$ reach simultaneously their race-variable-instrumented control-points, a race occurs, even if the cycle may never be closed by the remaining processes. In the following, we add not only race variables, but also *additional* locks, assuring that parts of a cycle do not already lead to a race; we call these locks *gate locks*. Adding new locks, however, needs to be done carefully so as not to change the behavior of the program, in particular, not to break Lemma 6.

We first define another (conceptual) use of locks, denoted *short-lived locks*. A process which is holding a short-lived lock has to first release it before trying any other lock. It is obvious to see that transforming a program by adding short-lived locks does not lead to more deadlocks.

A gate lock is a short-lived lock which is specially used to protect the access to race variables in a program. Since gate locks are short-lived, no new deadlocks will be introduced. Similar to the transformation in Definition 7, we still instrument with race variables at the static second lock points, but *also* wrap the access with locking/unlocking of the corresponding gate lock (there is one gate lock per Δ_C). However, we *pick one* of the processes in Δ_C which *only* accesses the race variable *without* the gate lock held. This transformation ensures that the picked process and exactly *one* of the other processes involved in a deadlock

cycle may reach the static second lock points at the same time, and thus a race occurs. That is, only the race between the process which could close the deadlock cycle and any *one* of the other processes involved in the deadlock will be triggered.

Observe that depending on the chosen process, the race checker may or may not report a race—due to the soundness of our approach, we are obviously interested in the best result, which is "no race detected". Therefore, we suggest to run the analysis with all processes to find the optimal result. Note that checks for different cycles and with different "special" processes for the gate lock-based instrumentation can easily be run in parallel or distributed. It is also possible to instrument a single program for the detection of multiple cycles: even though a lock statement can be a second lock point for multiple abstract locks, the transformations for each of them do not interfere with each other, and can be analyzed in a single race checker-run.

Theorem 1. *Given a program P, P^T is a transformed program of P instrumenting with race variables and gate locks, P is deadlock-free if P^T is race-free.*

6 Conclusion

We presented an approach to statically analyze multi-threaded programs by reducing the problem of deadlock checking to data race checking. The type and effect system statically over-approximates program points, where deadlocks may manifest themselves and instruments programs with additional variables to signal a race. Additional locks are added to avoid too many spurious false alarms. We show soundness of the approach, i.e., the program is deadlock free, if the corresponding transformed program is race free.

Numerous approaches have been investigated and implemented over the years to analyze concurrent and multi-threaded programs (cf. e.g. [18] for a survey of various static analyses). Not surprisingly, in particular approaches to prevent races [3] and/or deadlocks [8] have been extensively studied for various languages and are based on different techniques. (Type-based) analyses for race detection include [1] [10] [6] [19] [13] to name a few. Partly based on similar techniques, likewise for the prevention of deadlocks are [21] [14]. Static detection of potential deadlocks is a recurring topic: traditionally, a lock-analysis is carried out to discover whether the locks can be *ordered*, such that subsequent locks can only be acquired following that order [4]. Then, a deadlock is immediately ruled out as this construction precludes any "deadly embrace". The lock order may be specified by the user, or inferred [5]. To the best of our knowledge, our contribution is the first formulation of (potential) deadlocks in terms of data races. Due to the number of race variables introduced in the transformation, and assuming that race checking scales linearly in their number, we expect an efficiency comparable to explicit-state model checking.

In general, races are prevented not just by protecting shared data via locks; a good strategy is to avoid also shared data in the first place. The biggest challenge

for static analysis, especially when insisting on soundness of the analysis, is to achieve better approximations as far as the danger of shared, concurrent access is concerned. Indeed, the difference between an overly approximate analysis and one that is usable in practice lies not so much in obtaining more refined conditions for races as such, but to get a grip on the imprecision caused by aliasing, and the same applies to static deadlock prevention.

Future work A natural extension of our work would be an implementation of our type and effect system to transform concurrent programs written in e.g. C and Java. Complications in those languages like aliasing need to be taken into account, although results from a *may-alias* analysis could directly be consumed by our analysis. The potential blowup of source code-size through instantiation of function applications can be avoided by directly making use of context in the race-checker, instead of working on a source-based transformed program. As a first step, we intend to make our approach more applicable, to directly integrate the transformation-phase into *Goblint*, so that no explicit transformation of C programs needs to take place.

For practical applications, our restriction on a fixed number of processes will not fit every program, as will the required static enumeration of abstract cycle information. We presume that our approach will work best on code found e.g. in the realm of embedded system, where generally a more resource-aware programming style means that threads and other resources are statically allocated.

For lack of space, most of the proofs have been omitted here. Further details can be found in the accompanying technical report [17].

Acknowledgements. We are grateful for detailed discussion of *Goblint* to Kalmer Apinis, and Axel Simon, from TU München, Germany.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for Java. ACM Transactions on Programming Languages and Systems 28(2), 207–255 (2006)
2. Amtoft, T., Nielson, H.R., Nielson, F.: Type and Effect Systems: Behaviours for Concurrency. Imperial College Press (1999)
3. Beckman, N.E.: A survey of methods for preventing race conditions (May 2006), http://www.nelsbeckman.com/publications.html
4. Birrell, A.D.: An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Research Center (1989)
5. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: Object Oriented Programming: Systems, Languages, and Applications, OOPSLA 2002, Seattle, USA. ACM (November 2002); SIGPLAN Notices
6. Boyapati, C., Rinard, M.: A parameterized type system for race-free Java programs. In: Object Oriented Programming: Systems, Languages, and Applications, OOPSLA 2001. ACM (2001)

7. Coffman Jr., E.G., Elphick, M., Shoshani, A.: System deadlocks. Computing Surveys 3(2), 67–78 (1971)
8. Corbett, J.: Evaluating deadlock detection methods for concurrent software. IEEE Transactions on Software Engineering 22(3), 161–180 (1996)
9. Dijkstra, E.W.: Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven (1965); Reprinted in [12]
10. Flanagan, C., Freund, S.N.: Type inference against races. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 116–132. Springer, Heidelberg (2004)
11. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: ACM Conference on Programming Language Design and Implementation, PLDI. ACM (June 1993); SIGPLAN Notices 28(6)
12. Genyus, F.: Programming Languages. Academic Press (1968)
13. Grossman, D.: Type-safe multithreading in Cyclone. In: TLDI 2003: Types in Language Design and Implementation, pp. 13–25. ACM (2003)
14. Kobayashi, N.: Type-based information flow analysis for the π-calculus. Acta Informatica 42(4–5), 291–347 (2005)
15. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: ACM Conference on Programming Language Design and Implementation, PLDI, Ottawa, Ontario, Canada, pp. 308–319. ACM (June 2006)
16. Pun, K.I., Steffen, M., Stolz, V.: Deadlock checking by a behavioral effect system for lock handling. Journal of Logic and Algebraic Programming 81(3), 331–354 (2012); A preliminary version was published as University of Oslo, Dept. of Computer Science Technical, Report 404 (March 2011)
17. Pun, K.I., Steffen, M., Stolz, V.: Deadlock checking by data race detection. Technical report 421, University of Oslo, Dept. of Informatics (October 2012)
18. Rinard, M.: Analysis of multithreaded programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 1–19. Springer, Heidelberg (2001)
19. Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.: Automated type-based analysis of data races and atomicity. In: Ferrante, J., Padua, D.A., Wexelblat, R.L. (eds.) PPoPP 2005, pp. 83–94. ACM (2005)
20. Seidl, H., Vojdani, V.: Region analysis for race detection. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 171–187. Springer, Heidelberg (2009)
21. Vasconcelos, V., Martins, F., Cogumbreiro, T.: Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In: Beresford, A.R., Gay, S.J. (eds.) Pre-Proceedings of the Workshop on Programming Language Approaches to Concurrent and Communication-Centric Software, PLACES 2009. EPTCS, vol. 17, pp. 95–109 (2009)

Delta Modeling and Model Checking of Product Families*

Hamideh Sabouri[1] and Ramtin Khosravi[1,2]

[1] School of Computer Science and Electrical Engineering, College of Engineering, University of Tehran, Tehran, Iran
[2] School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

Abstract. Software product line engineering focuses on proactive reuse to reduce the cost of developing families of related systems. A recently proposed method to develop software product lines is delta modeling where a set of deltas specify modifications that should be applied to a core product to achieve other products. The main advantage of this technique is its modularity and flexibility. In this paper, we propose an approach to model check delta-oriented product lines. To this end, we transform a delta model to a corresponding annotated model where an application condition is associated to each statement. An application condition specifies the set of products that a statement is included in them. We present the semantics of the resulting model in form of a featured transition system where each transition is annotated with an application condition. Featured transition systems are supported by a variability-aware model checking technique that can be used to verify the annotated model.

1 Introduction

Software product line (SPL) engineering enables proactive reuse by developing a family of related products instead of developing individual products separately. To this end, the commonalities and differences between products should be modeled explicitly [1]. Feature models are widely used to model the variability in SPLs. A feature model is a tree of features containing mandatory and optional features as well as other constraints among them, e.g., mutual exclusion. A product is defined as a valid combination of features, and a family is the set of all possible products [2] (section 3).

Delta modeling [3] is a modular, flexible, and expressive modeling approach that is recently proposed to develop SPLs. In this approach, an SPL is represented by a core product and a set of deltas. Deltas represent modifications that must be applied to the core product to derive other products of the product line. Each delta has an application condition that specifies the feature configurations on which the modifications are applicable.

* This research was in part supported by a grant form IPM. (No.CS1390-4-02).

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 51–65, 2013.
DOI: 10.1007/978-3-642-40213-5_4,
© IFIP International Federation for Information Processing 2013

As SPL engineering is increasingly used in the development of mission-critical and safety-critical systems such as embedded systems [4], formal verification of software product lines is essential. Recently, a number of approaches has been proposed to deductively verify delta-oriented SPLs using theorem proving [5,6,7,8]. However, to the best of our knowledge, there is no approach applying model checking technique [9] to verify delta-oriented models of software product lines. On the other hand, in [10] model checking algorithm is adapted to be applicable to SPLs (which we refer to it as variability-aware model checking). The drawback of this approach is that it does not support modular modeling of SPLs as the underlying formal model of product family is annotative. In annotative modeling approaches, each transition/statement of the model is annotated by an application condition to indicate the feature combinations that enable the transition/statement. The model checker uses the annotations to determine the set of products that satisfy/violate a property.

In this paper, we propose an approach to model check delta-oriented models of product families. Due to the compositional nature of delta models, developing a technique to analyze the entire family is not reasonable. A simple strategy is to generate all products of an SPL and model check each individually. However, this strategy may involve redundant computations due to similarities among the products. An alternative is to take benefit from the variability-aware model checking techniques. To this end, we transform a delta model to a corresponding annotative model. The annotated model can then be verified using variability-aware model checking technique. We select Rebeca [11] for formal modeling which is an actor-based language with a formal foundation to model and verify concurrent and distributed systems (section 4.1). Recently, the ABS language [12] is developed to model the behavior of configurable and distributed systems using delta modeling. However it is not supported by a model checking tool yet. Therefore, we choose Rebeca which is supported by an accessible model checker Afra [13]. Due to modularity, object-based nature, and Java-like syntax of Rebeca, its adaptation to support delta modeling is straightforward (considering the work on delta-oriented modeling of object-oriented SPLs like Java [3]).

To provide an approach to model check delta-oriented product families, we extend Rebeca to support delta modeling (section 4.2). We also introduce annotations in Rebeca and define the semantics of annotated Rebeca models using featured transition systems (FTS) [10] to which the variability-aware model checking algorithm is applicable (section 4.3). In FTS, an application condition determines the feature combinations that enable a transition. Then, we propose a method to transform a delta model to a corresponding annotated model (section 5.1). We also justify the correctness of our proposed approach intuitively (section 5.2). There are two possible approaches to model check the resulting annotated Rebeca model: using its underlying FTS or generating a plain Rebeca model from it to use the existing model checker of Rebeca for verification (section 5.3).

The main contribution of our work is developing SPLs in a high-level and modular manner by employing delta-oriented modeling concept while taking

advantage of variability-aware model checking which is currently only applicable to annotative models of product families. The contributions of our paper can be summarized as follows:

– We extend Rebeca to support delta modeling which enables us to model families of actor systems.
– We extend Rebeca with annotations along with its semantics based on FTS to apply the existing variability-aware model checking techniques.
– We propose a method to transform delta models to annotated models which enables us to use existing techniques for annotated models of SPLs.

The Coffee Machine Example. We use Coffee Machine family as the running example in the paper. A coffee machine may serve coffee or tea or both. Adding extra milk and extra sugar may be supported optionally. The payment method of a coffee machine is either by coin or by card. □

2 Related Work

Several approaches has been proposed for formal modeling of SPLs using SMV [14], automata and transition based systems [15,10,16,17], process algebra [18], Petri nets [19], and Promela [20]. These approaches capture the behavior of the entire product family in a single model by including the variability information in it using annotations. Annotating a transition/statement with an application condition indicates the configurations that enable the transition/statement. In [18], an operator is added to CCS to specify alternative processes.

To model check annotated models, model checking technique has been adapted in [10] to verify featured transition systems. In FTS, an application condition determines the feature combinations that enable a transition. To explore the state space of an FTS, the track of products should be kept. Thus, a reachability relation is constructed while exploring the state space which is a set of pairs (s, px). Such couple indicates that state s is reachable by products in px.

Recently, a number of methods has been proposed to deductively verify delta-oriented SPLs. In [5], all derivable products of a delta-oriented SPL are verified incrementally using interactive theorem proving. In the first step, the core product is verified completely. Then, for each other product, the invalidated proofs and some new obligation proofs are proven. In [6], a family-based technique is proposed to reduce the cost of deductive verification for SPLs. For modular verification of software families, a Liskov principle is developed for delta-oriented programming in [7]. In [8], a transformational proof system is developed for delta-oriented programs which supports modular verification.

3 Background: Software Product Lines

Software product line engineering is a paradigm to develop software applications using platforms and mass customization. To this end, the commonalities and

differences between products should be modeled explicitly. Feature models [2] are widely used for this purpose. A feature model represents all possible products of a software product line in terms of features and relationships among them. A feature model is a tree of features that allows the *mandatory*, *optional*, *or*, and *xor* relationships among features. It also includes *requires* and *excludes* constraints between features. A product is derived from a feature set by making a decision to include/exclude each feature. A valid product conforms to the constraints that are specified in the feature model.

A configuration keeps track of including/excluding features. The root feature can be omitted in a configuration as it is included in all products. Having feature set $\mathcal{F}$ with n features, a configuration is defined as $c \in \{\mathit{true}, \mathit{false}, ?\}^n$ where $c_i = \mathit{true}/\mathit{false}$ represents inclusion/exclusion of the ith feature. The value '?' indicates that a feature is not included nor excluded yet. A configuration is *decided* if it does not contain any '?' values. In other words, a decision is made about inclusion/exclusion of all features. Otherwise, the configuration is *partial*.

Sets of products can be described using application conditions. An *application condition* φ is a propositional logic formula over a set of features $\mathcal{F}$, defined by $\varphi ::= \mathit{true} \mid f \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi$ where $f \in \mathcal{F}$.

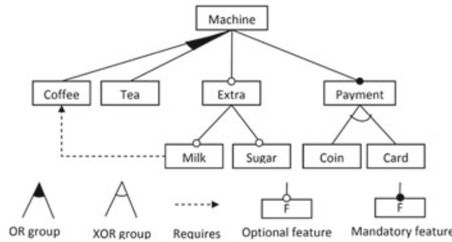


Fig. 1. The feature model of the coffee machine example

The Coffee Machine Example: Feature Model. The corresponding feature model of the coffee machine is depicted in Figure 1. Coffee and tea features have an *or* relationship implying that the machine serves one of these drinks at least. Adding extra milk and extra sugar is optional. However, when adding extra milk feature is available, the machine should be able to serve coffee because of the *requires* constraint between the milk and coffee features. Finally, the coin and card features have an *xor* relationship meaning that each product supports one and only one of them. A configuration that includes milk and excludes coffee is not a valid configuration. An example of a valid configuration is one that includes coffee, payment, and coin features and excludes the rest. □

4 Modeling Product Families in Rebeca

In this section, we describe two approaches to model product families in Rebeca: delta-oriented approach and annotative approach.

```
reactiveclass Controller() {
  knownrebecs {
    CoffeeMaker cm;
  }

  statevars {
    int cash, change, cost;
  }

  msgsrv initial() {
    cash, change = 0;
    cost = 1;
    self.receivePayment();
  }

  msgsrv nextOrder() {
    cm.makeCoffee(cash);
  }

  msgsrv receivePayment() {
    cash = ?(1,2,3);
    self.nextOrder();
  }

  msgsrv returnChange(int c) {
    change = cash - c;
    cash = 0;
    self.receivePayment();
  }
}
```

```
reactiveclass CoffeeMaker() {
  knownrebecs {
    Controller ctrl;
  }

  statevars {
    boolean addingCoffee;
    int cost;
  }

  msgsrv initial() {
    addingCoffee = false;
    cost = 1;
  }

  msgsrv makeCoffee(int cash) {
    addingCoffee = true;
    self.complete();
  }

  msgsrv complete() {
    ctrl.returnChange(cost);
    addingCoffee = false;
    cost = 1;
  }
}
```

Fig. 2. Rebeca model of a coffee machine

4.1 Rebeca

Rebeca is an actor-based language for modeling concurrent and distributed systems as a set of reactive objects which communicate via asynchronous message passing. A Rebeca model consists of a set of *reactive classes*. Each reactive class contains a set of *state variables* and a set of *message servers*. Message servers execute atomically, and process the receiving messages. The *initial* message server is used for initialization of state variables. A Rebeca model has a *main* part, where a fixed number of objects are instantiated from the reactive classes and execute concurrently. We refer to these objects as *rebecs*. The rebecs have no shared variable. Each rebec has a single thread of execution that is triggered by reading messages from an unbounded message queue. When a message is taken from the queue, its corresponding message server is invoked.

The Coffee Machine Example: Rebeca Model. Figure 2 shows the Rebeca model of a product from the coffee machine family that only includes coffee, payment, and coin features. In this model, the controller manages the payment as well as incoming orders and sends message to coffee maker accordingly. In the first step, the payment is received which can be at most three coins. Then, the drink may be ordered (the only option is coffee in this product). The controller is informed of the completion of serving the requested drink as well as the final cost so it can return the change and prepare for another order. Note that the cost of a drink may increase by adding extra milk or sugar in those products that support the corresponding features. □

4.2 Delta-Oriented Modeling in Rebeca

In delta modeling, an SPL is represented by a core product and a set of deltas. The core product is a valid product and deltas represent the modifications that should be applied to the core product to derive other products. In [21], the delta modeling technique is applied to object-oriented implementations of software product lines. Due to object-based nature of Rebeca language, we take a similar approach to [21] to introduce delta modeling in Rebeca.

Core Product A core product is a Rebeca model that captures the behavior of a product for a valid and decided feature configuration. Therefore, it contains a set of reactive classes and the main part where rebecs are instantiated.

The Coffee Machine Example: Core Product. We consider the product depicted in Figure 2 as the core product. As we mentioned earlier, it includes coffee, payment, and coin features and excludes other features. □

Deltas A delta represents the modification that must be applied to the core product to derive another product from the family. Deltas may add, remove, or modify reactive classes. Modifying a reactive class may add or remove known rebecs, state variables, and message servers. It may also change the behavior of an existing message server. Furthermore, an application condition is associated with each delta to specify the configurations for which the delta is applicable to the core product:

```
delta ⟨name⟩ [after ⟨delta names⟩] when ⟨application condition⟩{
    removes ⟨reactive class name⟩
    adds ⟨reactive class definition⟩
    modifies ⟨reactive class name⟩ {
        removes ⟨state variable/known rebec/message server name⟩
        adds ⟨state variable/known rebec/message server definition⟩
        modifies ⟨message server name⟩ ⟨message server definition⟩
    }
}
```

In the above description of a delta, the `when` clause represents the application condition and the `after` clause is used to specify the order of applying deltas.

The Coffee Machine Example: Deltas. A number of deltas that can be defined to describe a family of coffee machines are:

```
delta δ1 after δ3, δ4 when ¬coffee{
    removes CoffeeMaker
    modifies Controller {
        modifies nextOrder {I1}
    }
}
```

```
delta δ2 when tea {
    adds reactiveclass TeaMaker {...}
    modifies Controller {
        adds TeaMaker tm
        modifies nextOrder {I2}
    }
}
```

```
delta δ3 when milk {
    modifies CoffeeMaker {
        modifies makeCoffee {I3}
    }
}
```

```
delta δ4 when sugar {
    modifies CoffeeMaker {
        modifies makeCoffee {I4}
    }
}
```

Delta δ_1 defines the products without the coffee feature. Delta δ_2 adds the facility to serve tea. Deltas δ_3 and δ_4 are applied when adding extra milk and sugar are supported. □

Resolving Conflicts Conflicts among deltas may happen when they manipulate the same program entity in different ways. This leads to different implementations for one product when deltas are applied in different order. To achieve a unique product for a certain configuration, an order must be defined to apply deltas. A conflict-resolving delta δ_{ij} should be introduced to avoid conflict between two unordered deltas δ_i and δ_j. The application condition of δ_{ij} is the conjunction of the application conditions of δ_i and δ_j and δ_{ij} is applied later than both conflicting deltas (δ_{ij} has a higher priority than δ_i and δ_j).

The Coffee Machine Example: Conflicts. Deltas δ_1 and δ_2 are unordered and they are both modifying the implementation of the *nextOrder* message server. Another conflict exists between δ_3 and δ_4 which both modify the *makeCoffee* message server. Assume that in I_3 we add milk to coffee and increase the cost of the drink. In I_4 sugar is added and the cost is increased. When both features are available, the ultimate behavior of *makeCoffee* is determined by the delta that is applied later. To handle these issues, two conflict resolving deltas should be defined as δ_{12} and δ_{34}:

```
delta δ34 after δ3, δ4
when milk ∧ sugar {
   modifies CoffeeMaker {
      modifies makeCoffee {I5}
   }
}
```

```
delta δ12 after δ1, δ2
when ¬coffee ∧ tea {
   modifies Controller {
      modifies nextOrder {I6}
   }
}
```

In I_5 we add sugar and milk to coffee and determine the cost accordingly. □

Delta Model We define a delta model as a set of deltas along with their application conditions and priorities. A delta model is a triple $(\Delta, \Gamma, \prec)$ where

- Δ is a finite set of deltas,
- $\Gamma : \Delta \rightarrow \Phi_{\mathcal{F}}$ is function that associates an application condition with each delta,
- $\prec \subseteq \Delta \times \Delta$ is a partial order on Δ. $\delta_i \prec \delta_j$ states that δ_i should be applied before (not necessarily directly before) δ_j, when both deltas are applicable.

In the above definition, $\Phi_{\mathcal{F}}$ is the set of all possible application conditions over feature set $\mathcal{F}$. In our approach, we assume that we have an unambiguous delta model where all the conflicts among unordered deltas are resolved by defining appropriate conflict resolving deltas.

4.3 Annotated Rebeca Models

Syntax A fine-grained approach to represent variability in Rebeca model is to annotate the model with application conditions. We denote an annotation by $@\varphi$ where φ represents an application condition. In a Rebeca model, we may

annotate reactive classes, known rebecs, state variables, message servers, and statements (collectively referred to as *model entities*). Annotating an entity with an application condition specifies the set of products that include the entity. However, annotations on statements are enough to model SPLs. We can express annotations on other types of entities just by using annotations on statements, as shown below.

– A reactive class R annotated by $@\varphi$ is modeled by associating $@\varphi$ to every statement sending a message to a rebec r where r is an instance of R.
– A known rebec r annotated by $@\varphi$ is modeled by associating $@\varphi$ to every statement that sends message to r.
– A state variable v annotated by $@\varphi$ is modeled by associating $@\varphi$ to every statement that assigns to v or uses the value of v.
– A message server m annotated by $@\varphi$ is modeled by associating $@\varphi$ to every statement that sends the message m.

Hence, in the rest of this paper, we assume that only statements of message servers are annotated with application conditions when presenting the semantics of annotated Rebeca models.

The Coffee Machine Example: Annotations. We can annotate the Coffee maker reactive class as `@coffee reactiveclass CoffeeMaker`, to indicate that the reactive class is only available in the products supporting the coffee feature. An alternative way is to annotate the statement `cm.makeCoffee(cash)` with application condition `@coffee`. As a result, no message is sent to the coffee maker, thus it would be excluded from the model implicitly. □

Semantics The semantics of annotated Rebeca models can be described using a featured transition system (FTS). In [22], Rebeca semantics is described in form of labeled transition systems (LTS). In this work, we extend such LTS to an FTS to capture the notion of variability within an annotated Rebeca model. An FTS [10] is a transition system where the transitions are annotated using application conditions. Assuming that $\Phi_{\mathcal{F}}$ is the set of all possible application conditions over feature set $\mathcal{F}$, we define the semantics of an annotated Rebeca model as a featured transition system (S, I, A, T, γ) where

– S is a set of global states
– I is the initial state
– A is a set of actions (message servers)
– $T \subseteq S \times A \times S$ is a set of transitions
– $\gamma : T \rightarrow \Phi_{\mathcal{F}}$ associates an application condition to each transition

Each rebec has a local state composed of the values of its variables and the state of its queue: $\varsigma = \langle \mathcal{V}, q \rangle$. A Rebeca model consists of a number of rebecs executing concurrently. Thus, the global state is defined as the combination of the local states of all rebecs: $s = \prod \varsigma_i$. In the initial state, all of the queues only contain the *initial* message and all of the state variables have their default values. Message servers in Rebeca are executed in one atomic step, therefore an action corresponds to the execution of a message server. Variability in the behavior of

a model is realized through annotating the statements of message servers. The concept of variability is reflected in the semantics of Rebeca as follows.

We define message server m as a sequence of statements $\langle st_1; ...; st_n\rangle$ where φ_i is the application condition of st_i. To describe the semantics of variability in message servers easier, we consider that a sub-transition labeled by a sub-action from a sub-state to another, represents the execution of a single statement. In state s, execution of statement st_i has two possible outcomes. If φ_i does not hold, st_i is skipped without changing the local state of any rebecs. Otherwise when φ_i holds, execution of st_i may affect the local state of the currently executing rebec by changing the value of its state variables or putting a message in its queue (when the rebec sends a message to itself). Moreover, it may change the local state of other rebecs by putting messages in their queue. A possible path that denotes execution of m is: $s \xrightarrow{st_1,\varphi_1} \alpha_1 \xrightarrow{st_2,\neg\varphi_2} \alpha'_2 \xrightarrow{st_3,\varphi_3} ... \xrightarrow{st_n,\varphi_n} s'$.

Note that from each sub-state α_{i-1}, two sub-transitions with application conditions φ_i and $\neg\varphi_i$ are possible: $\alpha_i \xrightarrow{st_i,\varphi_i} \alpha_{i+1}$ and $\alpha_i \xrightarrow{st_i,\neg\varphi_i} \alpha'_{i+1}$. Due to atomic execution of message servers, we can compactly represent each execution path of m as a transition $t : s \xrightarrow{m,\gamma(t)} s'$ to denote removing message m from the queue of a rebec and executing it. Consequently, execution of message server m with n annotated statements leads to 2^n potential transitions from the current state s. The application condition of t is the conjunction of the application conditions of sub-transitions that constitute the path that t represents. For example, four possible transitions that represent execution of $m : \langle st_1, st_2\rangle$ are: $s \xrightarrow{m,\neg\varphi_1\wedge\neg\varphi_2} s'$, $s \xrightarrow{m,\neg\varphi_1\wedge\varphi_2} s''$, $s \xrightarrow{m,\varphi_1\wedge\neg\varphi_2} s'''$, and $s \xrightarrow{m,\varphi_1\wedge\varphi_2} s''''$.

4.4 Product Generation

Given a decided configuration c, a product can be derived from the model of the product family automatically. For this purpose, every application condition φ is evaluated by substituting all of its variables (each corresponds to a feature) by *true*/*false* based on c. By $c \vDash \varphi$ we denote that configuration c makes application condition φ *true*, otherwise $c \nvDash \varphi$.

Delta Model Having a core Rebeca model M_0 along with a delta model $D = (\Delta, \Gamma, \prec)$, a product with configuration c is obtained by applying every delta $\delta \in \Delta$ such that $c \vDash \Gamma(\delta)$ to M_0 considering the application order of deltas specified by $\prec$. The result is a plain Rebeca model for configuration c. We define $\Delta|_c \subseteq \Delta$ to contain all deltas that are applicable in c: $\Delta|_c = \{\delta_i \mid \delta_i \in \Delta \ \wedge \ c \vDash \delta_i\}$. Moreover, we assume that if $i < j$, either $\delta_i \prec \delta_j$ or δ_i and δ_j are unordered. Accordingly, we denote the model of the derived product corresponding to configuration c by $M_{\Delta|_c} = \delta_{c_k}(...(\delta_{c_1}(M_0))...)$ where $\Delta|_c = \{\delta_{c_1}, ..., \delta_{c_k}\}$.

Annotated Model The projection of an annotated Rebeca model R over a decided configuration c, denoted by $R|_c$, is a plain Rebeca model where the application conditions of every annotated reactive class, known rebec, state variable, message server, and statement are evaluated and those entities that their application condition does not hold for c are removed.

Note that the result of product generation (for delta-oriented or annotated models) may be a model that is not syntactically correct. For example, a rebec may send a message to another rebec that does not exists in the current configuration because it is removed by a delta (in case of delta modeling) or it is annotated with an application condition that does not hold (in case of annotative modeling). These inconsistencies can be detected by analyzing the model of product family statically. In this paper, we assume that every product that is derivable from the model of the product family is syntactically correct.

5 Model Checking Delta-Oriented Rebeca Models

A naive approach to model check SPLs is to generate the Rebeca model of every possible valid product (as we described in 4.4), then model check each product individually. This way, we lose the benefit of having commonalities among the products in the family. In this section, we propose an approach to transform a delta model to an annotated Rebeca model. Given the underlying FTS of an annotated Rebeca model, we can take benefit from variability-aware model checking technique proposed in [20] to model check delta-oriented actor systems. We can also use the late feature binding approach, proposed in [17] to handle variability in the model itself and use the existing model checker of Rebeca.

5.1 Transforming Deltas to Annotations

To transform delta model $D = (\Delta, \Gamma, \prec)$ to the corresponding annotated model, we modify the core model M_0 (which is a plain Rebeca model), according to the deltas defined in D. We assume that deltas with smaller identifiers has lower priority than deltas with greater identifiers. We start by changing the core model M_0 based on the modifications specified by delta δ_1 which results in the model M_1. Likewise, the model M_i is obtained by applying δ_i to M_{i-1}. Due to our earlier assumption on unambiguity and correctness of delta models, the transformation results in a unique annotated model. Moreover, it is not required to deal with cases such as removing an entity that does not exits.

We define a model to be the set of all entities that exist in it. An entity is a reactive class, message server, state variable, known rebec, or statement. Each entity e is represented by a pair $e = (n, d)$ where n is the name of the entity and d is its definition. For simplicity, we do not discuss the formal definition of d in this paper. Informally, known rebecs and state variables are defined by their types. A message server is defined by its parameters and its sequence of statements. Finally, a reactive class is defined by its set of state variables, known rebecs, and message servers. We assume unique names for every known rebec, state variable, and message servers. Unique names for these entities can be obtained by adding the name of their corresponding reactive class as a prefix to their names.

Having $\mathcal{F}$ as the feature set, we assume that function $\mathcal{A}_i : M_i \rightarrow \Phi_{\mathcal{F}}$ returns the application condition by which each entity is annotated in M_i. Function $\mathcal{A}_0$ returns *true* for all entities in M_0. By applying δ_{i+1} on M_i, we may add new

```
reactiveclass Controller() {
  knownrebecs {
    CoffeeMaker cm;
    @tea TeaMaker tm;
  }

  statevars {
    int req, cash, change, cost;
  }

  msgsrv initial() {
    cash, change = 0;
    cost = 1;
    self.receivePayment();
  }

  msgsrv nextOrder() {
    @¬(¬coffee ∧ tea) {
      @¬(¬coffee) {
        @¬tea { cm.makeCoffee(cash); }
        @tea { I2 }
      }
      @¬coffee { I1 }
    }
    @(¬coffee ∧ tea) { I6 }
  }

  msgsrv receivePayment() {
    cash = ?(1,2,3);
    self.nextOrder();
  }

  msgsrv returnChange(int c) {
    change = cash - c;
    cash = 0;
    self.receivePayment();
  }
}
```

```
@coffee reactiveclass CoffeeMaker() {
  knownrebecs {
    Controller ctrl;
  }

  statevars {
    boolean addingCoffee;
    int cost;
  }

  msgsrv initial() {
    addingCoffee = false;
    cost = 1;
  }

  msgsrv makeCoffee(int cash) {
    @¬(milk ∧ sugar) {
      @¬sugar {
        @¬milk {
          addingCoffee = true;
          self.complete();
        }
        @milk { I3 }
      }
      @sugar { I4 }
    }
    @(milk ∧ sugar) { I5 }
  }

  msgsrv complete() {
    ctrl.returnChange(cost);
    addingCoffee = false;
    cost = 1;
  }
}

@tea reactiveclass TeaMaker() {
  ...
}
```

Fig. 3. The annotated Rebeca model of the coffee machine family

entities to M_i or change the annotations of existing ones. We do not eliminate any entity from M_i as removing or modifying an entity is handled by updating its corresponding annotation. Thus, all the definitions of an entity specified by different deltas, coexist in the annotated model. These definitions are distinguished by their annotations. Having an unambiguous delta model, only one of these definitions is applicable for a specific configuration. The effect of delta δ_{i+1} on model M_i is captured in the annotated model as follows.

Adding an Entity Suppose δ_{i+1} adds entity $e = (n, d)$ to the model. Note that M_i may include one or more entities with the same name. This happens when for some $j < i$, δ_{j-1} adds an entity which is then removed by δ_j and added again in δ_{j+1}. In this case, we compare definition d with every definition for n in M_i. To handle adding an entity $e = (n, d)$ we consider the following cases:

- When e does not exist in M_i (there is no entity in M_i with name n), we add e to M_i and annotate it with its corresponding application condition: $\mathcal{A}_i(e) = \Gamma(\delta_{i+1})$.
- If d is different from all existing definitions in M_i (for the entities with name n), a new entity is added to M_i along with its annotation $\mathcal{A}_i(e) = \Gamma(\delta_{i+1})$.

– Otherwise, if there exists an entity $e' = (n, d')$ where d and d' are the same, we update the annotation of e' as: $\mathcal{A}_i(e') = \mathcal{A}_{i-1}(e') \vee \Gamma(\delta_{i+1})$.

We consider the definition of two message servers the same if they have the same parameters and the same sequence of statements. Two state variables/known rebecs with the same name are equal if they have the same type. Definitions of two reactive classes are equal if they have the same set of state variables, known rebecs, and message servers with identical definitions.

Removing an Existing Entity To handle removing an entity (n, d), we modify the annotation of all entities in M_i with the name n. For each $e_k = (n, d_k) \in M_i$, we conjunct its annotation with $\neg\Gamma(\delta_{i+1})$: $\mathcal{A}_i(e_k) = \mathcal{A}_{i-1}(e_k) \wedge \neg\Gamma(\delta_{i+1})$.

Note that by the above conjunction, we preserve the higher priority of δ_{i+1} over previously applied deltas as when $\Gamma(\delta_{i+1})$ holds, it makes the entire formula *false*. On the other hand, if we apply another delta δ_j later $(j > i)$ to add an entity with the same name and definition again, the new annotation will be $(\mathcal{A}_{i-1}(e_k) \wedge \neg\Gamma(\delta_{i+1})) \vee \Gamma(\delta_j)$. Consequently, the entity would be included in the model if a delta with higher priority adds the entity again.

Modifying the Implementation of a Message Server We assume that I_0 is the initial implementation of message server m. We consider δ_{m_i} to be the ith delta that changes the implementation of m. By applying $\delta_{m_{k+1}}$, the implementation of m changes from I_k to I_{k+1}. If $\delta_{m_{k+1}}$ specifies $I_{\delta_{k+1}}$ as the new implementation of m, then I_k is obtained by the sequential composition of I_k and $I_{\delta_{k+1}}$ annotated by $\neg\Gamma(\delta_{m_{k+1}})$ and $\Gamma(\delta_{m_{k+1}})$ respectively. This way, $I_{\delta_{k+1}}$ is executed only when $\Gamma(\delta_{m_{k+1}})$ holds. Consequently, that the body of message server m is defined recursively as: $I_{k+1} = \{\ @\neg\Gamma(\delta_{m_{k+1}})I_k;\ \ @\Gamma(\delta_{m_{k+1}})I_{\delta_{k+1}}\ \}$

The Coffee Machine Example: Transformation. Figure 3 shows the annotated Rebeca model after applying the deltas δ_2, δ_3, δ_4, δ_1, δ_{12}, and δ_{34}. We omit the details of each implementation I_i. □

5.2 Justification

In this section, we explain how our proposed approach may be justified intuitively. Proving the correctness of the approach formally is in our future agenda. Note that the following arguments only holds for decided configurations.

A model of a product family consists of a core Rebeca model M_0 along with its corresponding delta model D. By applying the applicable deltas to M_0 with respect to their predefined order, $M_{\Delta|_c}$ is obtained which is the model of the individual product with configuration c. The semantics of the resulting Rebeca model can be described using an LTS which we denote it by $[\![M_{\Delta|_c}]\!]$. We may transform the delta-oriented model of the product family to its corresponding annotative model M_a. The semantics of M_a, denoted by $[\![M_a]\!]^\star$, is defined using an FTS. Moreover, we may project M_a over a configuration c to obtain the Rebeca model of an individual product which $[\![M_a|_c]\!]$ represents its semantics.

The correctness of the proposed transformation can be established by proving that the underlying transition systems of $M_{\Delta|_c}$ and $M_a|_c$ are bisimilar: $[\![M_{\Delta|_c}]\!] \approx [\![M_a|_c]\!]$. According to the transformation rules, both transition systems have

the same set of message servers with same the behaviors as their action set. Therefore, starting from the initial states, both transition systems have the same set of enabled actions where taking each action implies the same behavior in both transition systems.

The variability-aware model checking is applicable on the FTS $[\![M_a]\!]^{\star}$. We can justify that such FTS includes the behavior of all products of the delta model by defining a refinement relation between two FTSs and proving that for every configuration c, $[\![M_{\Delta|_c}]\!] \sqsubseteq [\![M_a]\!]^{\star}$. Given two featured transition systems T_A and T_B, we say T_A refines T_B, denoted $T_A \sqsubseteq T_B$, if and only if there is a relation $\mathcal{R}$ between the states of their underlying transition systems such that $\mathcal{R}(s_{A_0}, s_{B_0})$. Moreover, If $\mathcal{R}(s_A, s_B)$ and there exist transition $t : s_A \xrightarrow{a} s'_A$ then there exists transition $t' : s_B \xrightarrow{a} s'_B$ such that $\mathcal{R}(s'_A, s'_B)$ and $\gamma(t) \Rightarrow \gamma(t')$. According to the proposed transformation rules and the presented semantics for annotated Rebeca models, $[\![M_{\Delta|_c}]\!] \sqsubseteq [\![M_a]\!]^{\star}$ can be justified intuitively for every configuration c. Note that every transition system (such as $[\![M_{\Delta|_c}]\!]$) is trivially an FTS with $\gamma(t) = \mathit{true}$ for every transition t.

5.3 Model Checking

We can use the model checking algorithm tailored for product families in [20] or use the late feature binding approach proposed in [17] to model check the resulting annotated model.

Variability Aware Model Checking In this approach, we use the underlying FTS of a Rebeca model to apply the variability-aware model checking technique developed to verify product families. This way, the model checker returns all the products that satisfy the given property along with counter-examples for those products that violate the property. To take benefit from such technique, we should alter the current compiler of Rebeca and adapt its model checker accordingly. This is one of our future work.

Traditional Model Checking To use the existing compiler and model checker of Rebeca, we transform the annotated model to a plain Rebeca model which handles variability within the model itself. Annotation $@\varphi$ for a statement s can be modeled using conditional statement `if(`φ`) s;` itself. To decide on including or excluding feature f, we use the late feature binding approach proposed in [17] where such decision is made just before using a feature to optimize the number of generated states. For this purpose, we model each feature f using an integer variable v_f where its value represent if it is included ($v_f = 1$), excluded ($v_f = 0$), or it is not included nor excluded ($v_f = -1$) yet. We decide on the value of a feature variable by adding `if(`v_f `== -1)` v_f = ?(0,1), just before its usage in the application condition of a statement. Such statement non-deterministically includes or excludes f when no decision is made for it yet. After transforming the annotated Rebeca model to a plain model, we model check it using existing model checker.

Result We have applied our approach on an extended version of the coffee machine example presented in this paper. We verified the annotated model against deadlock by replacing annotations with conditional statements, then

using the existing model checker of Rebeca. Deriving every product from the delta model and model checking it separately leads to 20,596 total states for all products. However, by transforming the delta model to an annotated model and applying the late binding technique, 1,360 states are generated for the entire family.

6 Conclusion

In this paper, we presented an approach to model product families in a high-level and modular manner, using delta-oriented modeling. To model check such a model, we transform it to a corresponding annotated model with its semantics defined by featured transition systems. Such transition systems can be verified using a variability-aware model checking technique to obtain the products that satisfy the given property. We may also apply the late feature binding technique to verify the entire family using existing model checking techniques. The result of applying our proposed approach on a coffee machine case study shows that it is more efficient to transform a delta model to an annotated one, then model check the entire family, rather than deriving each product from the delta model and verify them individually.

References

1. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc. (2005)
2. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)
3. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
4. Ebert, C., Jones, C.: Embedded software: Facts, figures, and future. Computer 42, 42–52 (2009)
5. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 61–75. Springer, Heidelberg (2011)
6. Thüm, T., Schaefer, I., Hentschel, M., Apel, S.: Family-based deductive verification of software product lines. In: Proc. Generative Programming and Component Engineering, GPCE 2012, pp. 11–20. ACM (2012)
7. Hähnle, R., Schaefer, I.: A liskov principle for delta-oriented programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 32–46. Springer, Heidelberg (2012)
8. Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E.B., Yu, I.C.: A transformational proof system for delta-oriented programming. In: Proc. Software Product Line Conference, SPLC 2012, vol. 2, pp. 53–60. ACM (2012)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
10. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proc. Int'l Conf. on Software Eng., ICSE 2010, pp. 335–344 (2010)

11. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inf. 63(4), 385–410 (2004)
12. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
13. Rebeca research group: Afra integrated verification environment for Rebeca, http://www.rebeca-lang.org
14. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. 41(1), 53–84 (2001)
15. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
16. Sabouri, H., Khosravi, R.: An effective approach for verifying product lines in presence of variability models. In: Proc. Software Product Lines, vol. 2, pp. 113–120 (2010)
17. Sabouri, H., Jaghoori, M.M., de Boer, F.S., Khosravi, R.: Scheduling and analysis of real-time software families. In: Proc. Computer Software and Applications. IEEE Computer Society (2012)
18. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008)
19. Muschevici, R., Clarke, D., Proença, J.: Feature Petri Nets. In: Proc. Software Product Lines, vol. 2, pp. 99–106 (2010)
20. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. International Journal on Software Tools for Technology Transfer, 1–24 (2012)
21. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. In: Proc. Generative Programming and Component Engineering, GPCE 2010, pp. 13–22 (2010)
22. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca. Acta Inf. 47(1), 33–66 (2010)

Lending Petri Nets and Contracts

Massimo Bartoletti, Tiziana Cimoli, and G. Michele Pinna

Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Cagliari, Italy

Abstract. Choreography-based approaches to service composition typically assume that, after a set of services has been found which correctly play the roles prescribed by the choreography, each service respects his role. Honest services are not protected against adversaries. We propose a model for contracts based on an extension of Petri nets, which allows services to protect themselves while still realizing the choreography. We relate this model with Propositional Contract Logic, by showing a translation of formulae into our Petri nets which preserves the logical notion of agreement, and allows for compositional verification.

1 Introduction

Many of today's human activities, from business and financial transactions, to collaborative and social applications, run over complex interorganizational systems, based on service-oriented computing (SOC) and cloud computing technologies. These technologies foster the implementation of complex software systems through the composition of basic building blocks, called *services*. Ensuring reliable coordination of such components is fundamental to avoid critical, possibly irreparable problems, ranging from economic losses in case of commercial activities, to risks for human life in case of safety-critical applications.

Ideally, in the SOC paradigm an application is constructed by dynamically discovering and composing services published by different organizations. Services have to *cooperate* to achieve the overall goals, while at the same time they have to *compete* to achieve the specific goals of their stakeholders. These goals may be conflicting, especially in case of mutually distrusted organizations. Thus, services must play a double role: while cooperating together, they have to protect themselves against other service's misbehavior (either unintentional or malicious).

The lack of precise guarantees about the reliability and security of services is a main deterrent for industries wishing to move their applications and business to the cloud [3]. Quoting from [3], "absent radical improvements in security technology, we expect that users will use contracts and courts, rather than clever security engineering, to guard against provider malfeasance".

Indeed, contracts are already a key ingredient in the design of SOC applications. A *choreography* is a specification of the overall behavior of an interorganizational process. This *global* view of the behavior is projected into a set of *local* views, which specify the behavior expected from each service involved in the whole process. The local views can be interpreted as the service contracts: if

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 66–82, 2013.
DOI: 10.1007/978-3-642-40213-5_5,
© IFIP International Federation for Information Processing 2013

the actual implementation of each service respects its contract, then the overall application must be guaranteed to behave correctly.

There are many proposals of formal models for contracts in the literature, which we may roughly divide into "physical" and "logical" models. Physical contracts take inspiration mainly from formalisms for concurrent systems (e.g. Petri nets [21], event structures [15,5], and various sorts of process algebras [8,9,10,12,16]), and they allow to describe the interaction of services in terms of response to events, message exchanges, *etc.* On the other side, logical contracts are typically expressed as formulae of suitable logics, which take inspiration and extend e.g. modal [1,14], intuitionistic [2,7], linear [2], deontic [18] logics to model high-level concepts such as promises, obligations, prohibitions, authorizations, *etc.*

Even though logical contracts are appealing, since they aim to provide formal models and reasoning tools for real-world Service Level Agreements, existing logical approaches have not had a great impact on the design of SOC applications. A reason is that there is no evidence on how to relate high-level properties of a contract with properties of the services which have to realize it. The situation is decidedly better in the realm of physical contracts, where the gap between contracts and services is narrower. Several papers, e.g. [9,10,11,16,21], address the issue of relating properties of a choreography with properties of the services which implement it (e.g. deadlock freedom, communication error freedom, session fidelity), in some cases providing automatic tools to project the choreography to a set services which correctly implements it.

A common assumption of most of these approaches is that services are *honest*, i.e. their behavior always adheres to the local view. For instance, if the local view takes the form of a behavioral type, it is assumed that the service is typeable, and that its type is a subtype of the local view. Contracts are only used in the "matchmaking" phase: once, for each local view projected from the choreography, a compliant service has been found, then all the contracts can be discarded.

We argue that the honesty assumption is not suitable in the case of interorganizational processes, where services may pursue their providers goals to the detriment to the other ones. For instance, consider a choreography which prescribes that a participant A performs action a (modeling e.g. "pay \$100 to B"), and that B performs b (e.g. "provide A with 5GB disk storage"). If both A and B are honest, then each one will perform its due action, so leading to a correct execution of the choreography. However, since providers have full control of the services they run, there is no authority which can force services to be honest. So, a malicious provider can replace a service validated w.r.t. its contract, with another one: e.g., B could wait until A has done a, and then "forget" to do b. Note that B may perform his scam while not being liable for a contract violation, since contracts have been discarded after validation.

In such competitive scenarios, the role of contracts is twofold. On the one hand, they must guarantee that their composition complies with the choreography: hence, in contexts where services are honest, the overall execution is correct. On the other hand, contracts must protect services from malicious ones: in the

example above, the contract of A must ensure that, if A performs a, then B will either do b, or he will be considered culpable of a contract violation.

In this paper, we consider physical contracts modeled as Petri nets, along the lines of [21]. In our approach we can both start from a choreography (modeled as a Petri net) and then obtain the local views by projection, as in [21], or start from the local views, i.e. the contracts published by each participant, to construct a choreography which satisfies the goals of everybody. Intuitively, when this happens the contracts admit an *agreement*.

A crucial observation of [6] is that if contracts admit an agreement, then some participant is not protected, and *vice-versa*. The archetypical example is the one outlined above. Intuitively, if each participant waits until someone else has performed her action, then everyone is protected, but the contracts do not admit an agreement because of the deadlock. Otherwise, if a participant does her action without waiting, then the contracts admit an agreement, but the participant who makes the first step is not protected. This is similar to the proof of impossibility of fair exchange protocols without a trusted third party [13].

To overcome this problem, we introduce *lending Petri nets* (in short, LPN). Roughly, an LPN is a Petri net where some places may give tokens "on credit". Technically, when a place gives a token on credit its marking will become negative. This differs from standard Petri nets, where markings are always nonnegative. The intuition is that if a participant takes a token on credit, then she is obliged to honour it — otherwise she is culpable of a contract violation.

Differently from the Petri nets used in [21], LPNs allow for modeling contracts which, at the same time, admit an agreement (more formally, *weakly terminate*) and protect their participants. LPNs preserve one of the main results of [21], i.e. the possibility of proving that an application respects a choreography, by only locally verifying the services which compose it. More precisely, we project a choreography to a set of local views, independently refine each of them, and be guaranteed then the composition of all refinements respects the choreography. This is stated formally in Theorem 8.

The other main contribution is a relation between the logical contracts of [7] and LPN contracts. More precisely, we consider contracts expressed in (a fragment of) Propositional Contract Logic (PCL), and we compile them into LPNs. Theorem 23 states that a PCL contract admits an agreement if and only if its compilation weakly terminates. Summing up, Theorem 24 states that one can start from a choreography represented as a logical contract, compile it to a physical one, and then use Theorem 8 to project it to a set services which correctly implement it, and which are protected against adversaries. Finally, Theorem 25 relates logical and physical characterizations of *urgent* actions, *i.e.* those actions which must be performed in a given state of the contract.

2 Nets

We briefly review Petri nets [19] and the token game. We consider Petri nets labeled on a set $\mathcal{T}$, and (perhaps a bit unusually) the labeling is also on places.

A *labeled Petri net* is a 5-tuple $\langle S, T, F, \Gamma, \Lambda \rangle$, where S is a set of *places*, and T is a set of *transitions* (with $S \cap T = \emptyset$), $F \subseteq (S \times T) \cup (T \times S)$ is the *flow relation*, and $\Gamma : S \to \mathcal{T}$, $\Lambda : T \to \mathcal{T}$ are partial *labeling function* for places and transitions, respectively. Ordinary (non labeled) Petri nets are those where the two labeling functions are always undefined (*i.e.* equal to $\bot$). We require that for each $t \in T$, $F(t, s) > 0$ for some place $s \in S$, i.e. a transition cannot happen *spontaneously*. Subscripts on the net name carry over the names of the net components. As usual, we define the *pre-set* and *post-set* of a transition/place: ${}^\bullet x = \{y \in T \cup S \mid F(y, x) > 0\}$ and $x^\bullet = \{y \in T \cup S \mid F(x, y) > 0\}$, respectively. These are extended to subsets of transitions/places in the obvious way.

A *marking* is a function m from places to natural numbers (*i.e.* a multiset over places), which represents the state of the system modeled by the net. A *marked* Petri net is a pair $N = (\langle S, T, F, \Gamma, \Lambda \rangle, m_0)$, where $\langle S, T, F, \Gamma, \Lambda \rangle$ is a labelled Petri net, and $m_0 : S \to \mathbb{N}$ is the *initial marking*.

The dynamic of a net is described by the execution of transitions at markings. Let N be a marked net (hereafter we will just call net a marked net). A transition t is enabled at a marking m if the places in the pre-set of t contains enough tokens (*i.e.* if m *contains* the pre-set of t). Formally, $t \in T$ is *enabled* at m if $m(s) \geq F(s, t)$ for all $s \in {}^\bullet t$. In this case, to indicate that the execution of t in m produces the new marking $m'(s) = m(s) - F(s, t) + F(t, s)$, we write $m\,[t\rangle\, m'$, and we call it a *step*[1]. This notion is lifted, as usual, to multisets of transitions.

The notion of step leads to that of *execution* of a net. Let $N = (\langle S, T, F, \Gamma, \Lambda \rangle, m_0)$ be a net, and let m be a marking. The *firing sequences* starting at m are defined as follows: (a) m is a firing sequence, and (b) if $m\,[t_1\rangle\, m_1 \cdots m_{n-1}\,[t_n\rangle\, m_n$ is a firing sequence and $m_n\,[t\rangle\, m'$ is a step, then $m\,[t_1\rangle\, m_1 \cdots m_{n-1}\,[t_n\rangle\, m_n\,[t\rangle\, m'$ is a firing sequence. A marking m is *reachable* iff there exists a firing sequence starting at m_0 leading to it. The set of reachable markings of a net N is denoted with $\mathsf{M}(N)$. A net $N = (\langle S, T, F, \Gamma, \Lambda \rangle, m_0)$ is *safe* when each marking $m \in \mathsf{M}(N)$ is such that $m(s) \leq 1$ for all $s \in S$.

A *trace* can be associated to each firing sequence, which is the word on $\mathcal{T}^*$ obtained by the firing sequence considering just the (labels of the) transitions and forgetting the markings: if $m_0\,[t_1\rangle\, m_1 \cdots m_{n-1}\,[t_n\rangle\, m_n$ is a firing sequence of N, the associated trace is $\Lambda(t_1 t_2 \ldots t_n)$. The trace associated to m_0 is the empty word ϵ. If the label of a transition is undefined then the associated word is the empty one. The traces of a net N are denoted with *Traces*(N).

A *subnet* is a net obtained by restricting places and transitions of a net, and correspondingly the flow relation and the initial marking. Let $N = (\langle S, T, F, \Gamma, \Lambda \rangle, m_0)$ be a net, and let $T' \subseteq T$. We define the subnet generated by T' as the net $N|_{T'} = (\langle S', T', F', \Gamma', \Lambda' \rangle, m_0')$, where $S' = \{s \in S \mid F(t, s) > 0$ or $F(s, t) > 0$ for $t \in T'\} \cup \{s \in S \mid m_0(s) > 0\}$, F' is the flow relation restricted to S' and T', Γ' is obtained by Γ restricting to places in S', Λ' is obtained by Λ restricting to transitions in T', and m_0' is obtained by m_0 restricting to places in S'.

[1] The word step is usually reserved to the execution of a subset of transitions, but here we prefer to stress the computational interpretation.

A net property (intuitively, a property of the system modeled as a Petri net) can be characterized in several ways, *e.g.* as a set of markings (states of the system). The following captures the intuition that, notwithstanding the state (marking) reached by the system, it is always possible to reach a state satisfying the property. A net N *weakly terminates in* a set of markings $\mathcal{M}$ iff $\forall m \in \mathsf{M}(N)$, there is a firing sequence starting at m and leading to a marking in $\mathcal{M}$. Hereafter, we shall sometimes say that N weakly terminates (without referring to any $\mathcal{M}$) when the property is not relevant or clear from the context.

We now introduce occurrence nets. The intuition behind this notion is the following: regardless how tokens are produced or consumed, an occurrence net guarantees that each transition can occur only once (hence the reason for calling them occurrence nets). We adopt the notion proposed by van Glabbeek and Plotkin in [22], namely 1-occurrence nets. For a multiset M, we denote by $[\![M]\!]$ the multiset defined as $[\![M]\!](a) = 1$ if $M(a) > 0$ and $[\![M]\!](a) = 0$ otherwise. A *state* of a net $N = (\langle S, T, F, \Gamma, \Lambda \rangle, m_0)$ is any finite multiset X of T such that the function $m_X : S \to \mathbb{Z}$ given by $m_X(s) = m_0(s) + \sum_{t \in T} X(t) \cdot (F(t, s) - F(s, t))$, for all $s \in S$, is a reachable marking of the net. We denote by $\mathsf{St}(N)$ the states of N. A state contains (in no order) all the occurrence of the transitions that have been fired to reach a marking. Observe that a trace of a net is a suitable linearization of the elements of a state X. We use the notion of state to formalize occurrence nets. An *occurrence net* $O = (\langle S, T, F, \Gamma, \Lambda \rangle, m_0)$ is a net where each state is a set, i.e. $\forall X \in \mathsf{St}(N).\ X = [\![X]\!]$.

A net is *correctly labeled* iff $\forall s. \forall t, t' \in {}^{\bullet}s.\ \Gamma(s) \neq \bot \implies \Lambda(t) = \Lambda(t') = \Gamma(s)$. Intuitively, this requires that all the transitions putting a token in a labeled place represent the same action.

3 Nets with Lending Places

We now relax the conditions under which transitions may be executed, by allowing a transition to consume tokens from a place s even if the s does not contain enough tokens. Consequently, we allow markings with negative numbers. When the number of tokens associated to a place becomes negative, we say that they have been done *on credit*. We do not permit this to happen in all places, but only in the *lending* places (a subset $\mathcal{L}$ of S). Lending places are depicted with a double circle.

Definition 1. *A* lending *Petri net (LPN) is a triple* $(\langle S, T, F, \Gamma, \Lambda \rangle, m_0, \mathcal{L})$ *where* $(\langle S, T, F, \Gamma, \Lambda \rangle, m_0)$ *is a marked Petri net, and* $\mathcal{L} \subseteq S$ *is the set of* lending *places.*

Example 1. Consider the LPN N_1 in Fig. 1. The places p_2 and p_4 are lending places. The set of labels of the transitions is $\mathcal{T} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, and the set of labels of the places is $\mathcal{G} = \mathcal{T}$. The labeling is $\Gamma(p_1) = \mathsf{c}, \Gamma(p_2) = \mathsf{a}$ and $\Gamma(p_4) = \Gamma(p_3) = \mathsf{b}$ (the place p_0 is unlabeled).

The notion of step is adapted to take into account this new kind of places. Let N be an LPN, let t be a transition in T, and let m be a marking. We say that

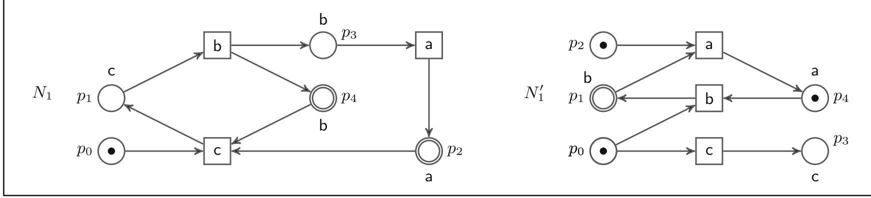


Fig. 1. Two lending Petri nets

t is *enabled* at m iff $\forall s \in {}^\bullet t.\ m(s) \leq 0 \implies s \in \mathfrak{L}$. The evolution of N is defined as before, with the difference that the obtained marking is now a function from places to $\mathbb{Z}$ (instead of $\mathbb{N}$). This notion matches the intuition behind of lending places: we allow a transition to be executed even when some of the transitions that are a pre-requisite have not been executed yet.

Definition 2. *Let m be a reachable marking of an LPN N. We say that m is* honored *iff $m(s) \geq 0$ for all places s of N.*

An honored firing sequence is a firing sequence where the final marking is honored. Note that if the net has no lending places, then all the reachable markings are honored.

Example 2. In the net of Ex. 1, the transition c is enabled even though there are no tokens in the places p_2 and p_4 in its pre-set, as they are lending places. The other transitions are not enabled, hence at the initial marking only c may be executed (on credit). After firing c, only b can be executed. This results in putting one token in p_3 and one in p_4, hence giving back the one taken on credit. After this, only a can be executed. Upon firing c, b and a, the marking is honored. The net is clearly a (correctly labeled) occurrence net.

We now introduce a notion of composition of LPNs. The idea is that the places with a label are places in an *interface* of the net (though we do not put any limitation on such places, as done instead *e.g.* in [21]) and they never are initially marked. The labelled transitions of a net are connected with the places bearing the same label of the other.

Definition 3. *Let $N = (\langle S, T, F, \Gamma, \Lambda\rangle,\ m_0, \mathfrak{L})$ and $N' = (\langle S', T', F', \Gamma', \Lambda'\rangle,$ $m'_0,\ \mathfrak{L}')$ be two LPNs. We say that N, N' are* compatible *whenever (a) they have the same set of labels, (b) $S \cap S' = \emptyset$, (c) $T \cap T' = \emptyset$, (d) $m_0(s) = 1$ implies $\Gamma(s) = \bot$, and (e) $m'_0(s') = 1$ implies $\Gamma'(s') = \bot$. If N and N' are compatible, their composition $N \oplus N'$ is the LPN $(\langle \hat{S}, T \cup T', \hat{F}, \hat{\Gamma}, \hat{\Lambda}\rangle, \hat{m}_0,\ \hat{\mathfrak{L}})$ in Fig. 2.*

The underlying idea of LPN composition is rather simple: the sink places in a net bearing a label of a transition of the other net are removed, and places and transitions with the same label are connected accordingly (the removed sink places have places with the same label in the other net). All the other ingredients

$$
\begin{array}{lcl}
\hat{S} & = & (S \setminus \{s \in S \mid \Gamma(s) \in \Lambda'(T') \text{ and } s^\bullet = \emptyset\}) \cup \\
 & & (S' \setminus \{s' \in S' \mid \Gamma'(s') \in \Lambda(T) \text{ and } s'^\bullet = \emptyset\}) \\
\hat{F}(\hat{s},\hat{t}) & \iff & \big(\hat{s} = s_1 \in S \ \wedge \ \hat{t} = t_1 \in T \ \wedge \ F(s_1,t_1)\big) \\
 & \vee & \big(\hat{s} = s_2 \in S' \ \wedge \ \hat{t} = t_2 \in T' \ \wedge \ F'(s_2,t_2)\big) \\
\hat{F}(\hat{t},\hat{s}) & \iff & \big(\hat{s} = s_1 \in S \ \wedge \ \hat{t} = t_1 \in T \ \wedge \ F(t_1,s_1)\big) \\
 & \vee & \big(\hat{s} = s_2 \in S' \ \wedge \ \hat{t} = t_2 \in T' \ \wedge \ F'(t_2,s_2)\big) \\
 & \vee & \big(\hat{s} = s \in S \ \wedge \ \hat{t} = t' \in T' \ \wedge \ \Lambda'(t') = \Gamma(s) \neq \bot\big) \\
 & \vee & \big(\hat{s} = s' \in S' \ \wedge \ \hat{t} = t \in T \ \wedge \ \Lambda(t) = \Gamma'(s') \neq \bot\big) \\
\hat{\Gamma}(\hat{s}) & = & \begin{cases} \Gamma(s_1) & \textit{if } \hat{s} = s_1 \in S \\ \Gamma'(s_2) & \textit{if } \hat{s} = s_2 \in S' \end{cases} \\
\hat{\Lambda}(\hat{t}) & = & \begin{cases} \Lambda(t_1) & \textit{if } \hat{t} = t_1 \in T \\ \Lambda'(t_2) & \textit{if } \hat{t} = t_2 \in T' \end{cases} \\
\hat{m}_0(\hat{s}) & = & \begin{cases} 1 & \text{if } \hat{s} = s_1 \in S \text{ and } m_0(s_1) = 1, \text{ or } \hat{s} = s_2 \in S' \text{ and } m_0'(s_2) = 1 \\ 0 & \text{otherwise} \end{cases} \\
\hat{\mathcal{L}} & = & (\mathcal{L} \cup \mathcal{L}') \cap \hat{S}
\end{array}
$$

Fig. 2. Composition of two LPNs

of the compound net are trivially inherited from the components. Observe that, when composing two compatible nets N and N' such that $\Gamma(S) \cap \Gamma'(S') = \emptyset$, we obtain the disjoint union of the two nets. Further, if the common label $a \in \Gamma(S) \cap \Gamma'(S')$ is associated in N to a place s with empty post-set and in N' to a place s' with empty post-set (or *vice versa*) and the labelings are injective, we obtain precisely the composition defined in [21]. If the components N and N' may satisfy some properties (sets of markings $\mathcal{M}$ and $\mathcal{M}'$), the compound net $N \oplus N'$ may satisfy the compound property (which is the set of markings $\hat{\mathcal{M}}$ obtained obviously from $\mathcal{M}$ and $\mathcal{M}'$).

Example 3. Consider the nets in Fig. 3. Net N fires a after b has been performed; dually, net N' waits for b before firing a. These nets model two participants which protect themselves by waiting the other one to make the first step (the properties being that places p_3 and p_3', respectively, are not marked). Clearly, no agreement is possible in this scenario. This is modelled by the deadlock in the composition $N \oplus N'$, where neither transitions a nor b can be fired. Consider now the LPN N'', which differs from N only for the lending place p_1''. This models a participant which may fire a on credit, under the *guarantee* that the credit will be eventually honoured by the other participant performing b (hence, the participant modeled by N'' is still protected), and the property is then place p_3'' unmarked and p_1'' with a non negative marking. The composition $N'' \oplus N'$ weakly terminates wrt the above properties, because transition a can take a token on credit from p_1'', and then transition b can be fired, so honouring the debit in p_1''.

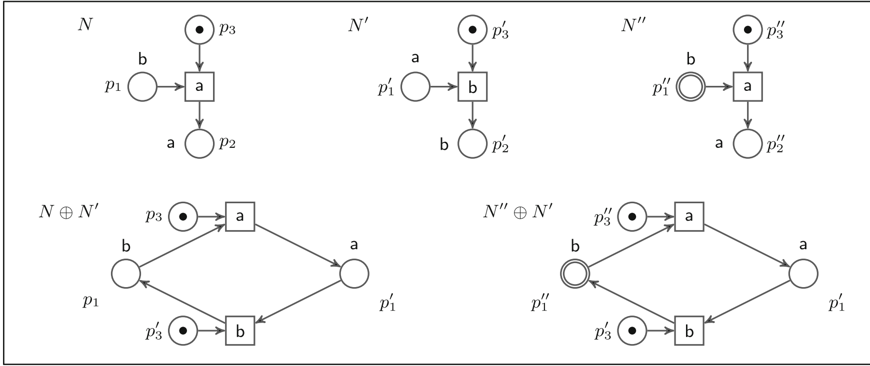


Fig. 3. Three LPNs (top) and their pairwise compositions (bottom)

The operation $\oplus$ is clearly associative and commutative.

Proposition 4. *Let N_1, N_2 and N_3 be three compatible LPNs. Then, $N_1 \oplus N_2 = N_2 \oplus N_1$ and $N_1 \oplus (N_2 \oplus N_3) = (N_1 \oplus N_2) \oplus N_3$.*

The composition $\oplus$ does not have the property that, in general, considering only the transitions of one of the components, we obtain the LPN we started with, *i.e.* $(N_1 \oplus N_2)|_{T_i} \neq N_i$. This is because the number of places with labels increases and new arcs may be added, and these places are not *forgotten* when considering the subnet generated by T_i. However these added places are not initially marked, hence it may be that the nets have the same traces.

Definition 5. *Let N and N' two LPNs on the same sets of labels. We say that N* approximates *N' ($N \lesssim N'$) iff $Traces(N) \subseteq Traces(N')$. We write $N \sim N'$ when $N \lesssim N'$ and $N' \lesssim N$.*

Proposition 6. *For two compatible LPNs N_1, N_2, $N_i \sim (N_1 \oplus N_2)|_{T_i}$, $i = 1, 2$.*

Following [21] we introduce a notion of refinement (called *accordance* in [21]) between two LPNs. We say that M (with a property $\mathcal{M}_M$) is a *strategy* for an LPN N (with a property $\mathcal{M}$) if $N \oplus M$ is weakly terminating. With $\mathcal{S}(N)$ we denote the set of all strategies for N. In the rest of the paper we assume that properties are always specified, even when not done explicitly.

Definition 7. *An LPN N'* refines *N if $\mathcal{S}(N') \supseteq \mathcal{S}(N)$.*

Observe that if N' refines N and N weakly terminates, then N' weakly terminates as well.

If a weakly terminating LPN N is obtained by composition of several nets, *i.e.* $N = \bigoplus_i N_i$, we can ask what happens if there is an N'_i which refines N_i, for each i. The following theorem gives the desired answer.

Theorem 8. *Let $N = \bigoplus_i N_i$ be a weakly terminating LPN, and assume that N'_i refines N_i, for all i. Then, $N' = \bigoplus_i N'_i$ is a weakly terminating LPN.*

The theorem above gives a compositional criterion to check weak termination of a SOC application. One starts from an abstract specification (e.g. a choreography), projects it into a set of local views, and then refines each of them into a service implementation. These services can be verified independently (for refinement), and it is guaranteed that their composition still enjoys weak termination.

We now define, starting from a marking m, which actions may be performed immediately after, while preserving the ability to reach an honored marking. We call these actions *urgent*.

Definition 9. *For an LPN N and marking m,we say* a urgent at *m iff there exists a firing sequence $m\,[t_1\rangle \cdots [t_n\rangle\, m_n$ with $\Lambda(t_1) = \mathsf{a}$ and m_n honored.*

Example 4. Consider the nets in Ex. 3. In $N'' \oplus N'$ the only urgent action at the initial marking is a, while b is urgent at the marking where p_1' is marked. In N'' there are no urgent actions at the initial marking, since no honored marking is reachable. In the other nets (N, N', $N \oplus N'$) no actions are urgent in the initial marking, since these nets are deadlocked.

4 Physical Contracts

We now present a model for physical contracts based on LPNs. Let $\mathsf{a}, \mathsf{b}, \ldots \in \mathcal{T}$ be *actions*, performed by *participants* $\mathsf{A}, \mathsf{B}, \ldots \in \mathit{Part}$. We assume that actions may only be performed once. Hence, we consider a subclass of LPNs, namely occurrence nets, where all the transitions with the same label are mutually exclusive. A physical contract is an LPN, together with a set $\mathcal{A}$ of participants bound by the contract, a mapping π from actions to participants, and a set Ω modeling the states where all the participant in $\mathcal{A}$ are satisfied.

Definition 10. *A* contract net $\mathcal{D}$ *is a tuple $(O, \mathcal{A}, \pi, \Omega)$, where O is an occurrence LPN $(\langle S, T, F, \Gamma, \Lambda\rangle, m_0, \mathcal{L})$ labeled on $\mathcal{T}$, $\mathcal{A} \subseteq \mathit{Part}$, $\pi : \mathcal{T} \to \mathit{Part}$, $\Omega \subseteq \wp(\mathcal{T})$ is the set of goals of the participants, and where:*

(a) $\forall s \in S.\ (m_0(s) = 1 \implies {}^\bullet s = \emptyset \ \wedge\ \Gamma(s) = \bot) \ \wedge\ (s \in \mathcal{L} \implies \Gamma(s) \in \mathcal{T})$,
(b) $\forall t \in T.\ (\forall s \in t^\bullet.\ \Lambda(t) = \Gamma(s)) \ \wedge\ (\exists s \in {}^\bullet t.\ s \notin \mathcal{L})$,
(c) $\forall t, t' \in T.\ \Lambda(t) = \Lambda(t') \implies \exists s \in {}^\bullet t \cap {}^\bullet t'.\ m_0(s) = 1$,
(d) $\pi(\Lambda(T)) \subseteq \mathcal{A}$.

The last constraint models the fact that only the participants in $\mathcal{A}$ may perform actions in $\mathcal{D}$.

Given a state X of the component O of $\mathcal{D}$, the reached marking m tells us which actions have been performed, and which tokens have been taken on credit. The *configuration* $\mu(m)$ associated to a marking m is the pair (C, Y) defined as:

- $C = \{\mathsf{a} \in \mathcal{T} \mid \exists s \in S.\ \{s\} = \bigcap_{t \in T}\{{}^\bullet t \mid \Lambda(t) = \mathsf{a}\}$ *and* $m(s) = 0\}$, and
- $Y = \{\mathsf{a} \in \mathcal{T} \mid \exists s \in S.\ \mathsf{a} = \Gamma(s)$ *and* $m(s) < 0\}$

The first component is the set of the labels of the transitions in X. The marking m is honored whenever the second component of $\mu(m)$ is empty.

We now state the conditions under which two contract nets can be composed. We require that an action can be performed only by one of the components (the other may *use* the tokens produced by the execution of such action).

Definition 11. *Two contracts nets* $\mathcal{D} = (O, \mathcal{A}, \pi, \Omega)$ *and* $\mathcal{D}' = (O', \mathcal{A}', \pi', \Omega')$ *are* compatible *whenever* $O \oplus O'$ *is defined and* $\mathcal{A} \cap \mathcal{A}' = \emptyset$.

The composition of $\mathcal{D}$ and $\mathcal{D}'$ is then the obvious extension of the one on LPNs:

Definition 12. *Let* $\mathcal{D} = (O, \mathcal{A}, \pi, \Omega)$ *and* $\mathcal{D}' = (O', \mathcal{A}', \pi', \Omega')$ *be two compatible contract nets. Then* $\mathcal{D} \oplus \mathcal{D}' = (O \oplus O', \mathcal{A} \cup \mathcal{A}', \pi \circ \pi', \Omega'')$ *where* $\Omega'' = \{X \cup X' \mid X \in \Omega, X' \in \Omega'\}$.

We lift the notion of weak termination to contract nets $\mathcal{D} = (O, \mathcal{A}, \pi, ok, \Omega)$. The set of markings obtained by Ω is $\mathcal{M}_\Omega = \{m \in \mathsf{M}(O) | \, \mu(m) = (C, \emptyset),\ C \in \Omega\}$. We say that $\mathcal{D}$ weakly terminates w.r.t. Ω when O weakly terminates w.r.t. $\mathcal{M}_\Omega$.

We also extend to contract nets the notion of urgent actions given for LPNs (Def. 9). Here, the set of urgent actions $\mathcal{U}_\mathcal{D}^C$ is parameterized by the set C of actions already performed.

Definition 13. *Let* $\mathcal{D}$ *be a contract net, and let* $C \subseteq \mathcal{T}$. *We define:*

$$\mathcal{U}_\mathcal{D}^C = \{\mathsf{a} \in \mathcal{T} \mid \exists Y \subseteq \mathcal{T}.\ \exists m.\ \mu(m) = (C, Y) \quad \wedge\ \mathsf{a} \quad \textit{is urgent at} \quad m\}$$

Example 5. Interpret the LPN N_1' in Fig. 1 as a contract net where the actions a, b, c are associated, respectively, to participants A, B, and C, and Ω is immaterial. Then, a and c are urgent at the initial marking, whereas b is not (the token borrowed from p_1 cannot be given back). In the state where a has been fired, only b is urgent; in the state where c has been fired, no actions are urgent.

5 Logical Contracts

In this section we briefly review Propositional Contract Logic (PCL [7]), and we exploit it to model contracts. PCL extends intuitionistic propositional logic IPC with a connective $\twoheadrightarrow$, called *contractual implication.* Intuitively, a formula $\mathsf{b} \twoheadrightarrow \mathsf{a}$ implies a not only when b is true, like IPC implication, but also in the case that a "compatible" formula, e.g. $\mathsf{a} \twoheadrightarrow \mathsf{b}$, holds. PCL allows for a sort of "circular" assume-guarantee reasoning, hinted by $(\mathsf{b} \twoheadrightarrow \mathsf{a}) \wedge (\mathsf{a} \twoheadrightarrow \mathsf{b}) \rightarrow \mathsf{a} \wedge \mathsf{b}$, which is a theorem in PCL. We assume that the prime formulae of PCL coincide with the atoms in $\mathcal{T}$. PCL formulae, ranged over greek letters $\varphi, \varphi', \ldots$, are defined as:

$$\varphi ::= \ \bot \ \mid \ \top \ \mid \ \mathsf{a} \ \mid \ \neg\varphi \ \mid \ \varphi \vee \varphi \ \mid \ \varphi \wedge \varphi \ \mid \ \varphi \rightarrow \varphi \ \mid \ \varphi \twoheadrightarrow \varphi$$

Two proof systems have been presented for PCL : a sequent calculus [7], and an equivalent natural deduction system [4], the main rules of which are shown

$$\frac{\Delta \vdash \psi}{\Delta \vdash \varphi \twoheadrightarrow \psi}\ (\twoheadrightarrow\text{I1}) \qquad \frac{\Delta \vdash \varphi' \twoheadrightarrow \psi' \quad \begin{array}{c}\Delta, \varphi \vdash \varphi' \\ \Delta, \psi' \vdash \varphi \twoheadrightarrow \psi\end{array}}{\Delta \vdash \varphi \twoheadrightarrow \psi}\ (\twoheadrightarrow\text{I2}) \qquad \frac{\begin{array}{c}\Delta \vdash \varphi \twoheadrightarrow \psi \\ \Delta, \psi \vdash \varphi\end{array}}{\Delta \vdash \psi}\ (\twoheadrightarrow\text{E})$$

Fig. 4. Natural deduction for PCL (rules for $\twoheadrightarrow$)

in Fig. 4. Provable formulae are contractually implied, according to rule ($\twoheadrightarrow$I1). Rule ($\twoheadrightarrow$I2) provides $\twoheadrightarrow$ with the same weakening properties of $\rightarrow$. The crucial rule is ($\twoheadrightarrow$E), which allows for the elimination of $\twoheadrightarrow$. Compared to the rule for elimination of $\rightarrow$ in IPC, the only difference is that in the context used to deduce the antecedent φ, rule ($\twoheadrightarrow$E) also allows for using as hypothesis the consequence ψ. The decidability of the provability relation of PCL has been proved in [7], by exploiting the cut elimination property enjoyed by the sequent calculus.

To model contracts, we consider the Horn fragment of PCL, which comprises atoms, conjunctions, and non-nested (intuitionistic/contractual) implications.

Definition 14. *A* PCL *contract is a tuple* $\langle \Delta, \mathcal{A}, \pi, \Omega \rangle$*, where* Δ *is a Horn* PCL *theory,* $\mathcal{A} \subseteq$ *Part,* $\pi : \mathcal{T} \rightarrow$ *Part associates each atom with a participant, and* $\Omega \subseteq \wp(\mathcal{T})$ *is the set of goals of the participants.*

The component $\mathcal{A}$ of $\mathcal{C}$ contains the participants which can promise to do something in $\mathcal{C}$. Consequently, we shall only consider PCL contracts such that if $\alpha \circ \mathsf{a} \in \Delta$, for $\circ \in \{\rightarrow, \twoheadrightarrow\}$, then $\pi(\mathsf{a}) \in \mathcal{A}$.

Example 6. Suppose three kids want to play together. Alice has a toy airplane, Bob has a bike, and Carl has a toy car. Each of the kids is willing to share his toy, but they have different constraints: Alice will lend her airplane only *after* Bob has allowed her ride his bike; Bob will lend his bike after he has played with Carl's car; Carl will lend his car if the other two kids promise to eventually let him play with their toys. Let $\pi = \{\mathsf{a} \mapsto \mathsf{A}, \mathsf{b} \mapsto \mathsf{B}, \mathsf{c} \mapsto \mathsf{C}\}$. The kids contracts are modeled as follows: $\langle \mathsf{b} \rightarrow \mathsf{a}, \{\mathsf{A}\}, \pi, \{\{\mathsf{b}\}\}\rangle$, $\langle \mathsf{c} \rightarrow \mathsf{b}, \{\mathsf{B}\}, \pi, \{\{\mathsf{c}\}\}\rangle$, and $\langle (\mathsf{a} \wedge \mathsf{b}) \twoheadrightarrow \mathsf{c}, \{\mathsf{C}\}, \pi, \{\{\mathsf{a}, \mathsf{b}\}\}\rangle$.

A contract admits an *agreement* when all the involved participants can reach their goals. This is formalized in Def. 15 below.

Definition 15. *A* PCL *contract admits an* agreement *iff* $\exists X \in \Omega.\ \Delta \vdash \bigwedge X$.

We now define composition of PCL contracts. If $\mathcal{C}'$ is the contract of an adversary of $\mathcal{C}$, then a naïve composition of the two contracts could easily lead to an attack, e.g. when Mallory's contract says that Alice is obliged to give him her airplane. To prevent from such kinds of attacks, contract composition is a partial operation. We do *not* compose contracts which bind the same participant, or which disagree on the association between atoms and participants.

$$\frac{}{\varepsilon \in [\![\Delta]\!]}\,(\varepsilon) \qquad \frac{\alpha \to \mathsf{a} \in \Delta \quad \sigma \in [\![\Delta]\!] \quad \overline{\alpha} \subseteq \overline{\sigma}}{\sigma\,\mathsf{a} \in [\![\Delta]\!]}\,(\to) \qquad \frac{\alpha \twoheadrightarrow \mathsf{a} \in \Delta \quad \sigma \in [\![\Delta, \mathsf{a}]\!] \quad \overline{\alpha} \subseteq \overline{\sigma}}{\sigma \mid \mathsf{a} \subseteq [\![\Delta]\!]}\,(\twoheadrightarrow)$$

Fig. 5. Proof traces of Horn PCL

Definition 16. *Two* PCL *contracts* $\mathcal{C} = \langle \Delta, \mathcal{A}, \pi, \Omega \rangle$ *and* $\mathcal{C}' = \langle \Delta', \mathcal{A}', \pi', \Omega' \rangle$ *are* compatible *whenever* $\mathcal{A} \cap \mathcal{A}' = \emptyset$, *and* $\forall \mathsf{A} \in \mathcal{A} \cup \mathcal{A}'.\ \pi^{-1}(\mathsf{A}) = \pi'^{-1}(\mathsf{A})$. *If* $\mathcal{C}$, $\mathcal{C}'$ *are compatible, the contract* $\mathcal{C} \mid \mathcal{C}' = \langle \Delta \cup \Delta', \mathcal{A} \cup \mathcal{A}', \pi \circ \pi', \Omega \mid \Omega' \rangle$, *where* $\Omega \mid \Omega' = \{X \cup X' \mid X \in \Omega,\ X' \in \Omega'\}$, *is their composition.*

Example 7. The three contracts in Ex. 6 are compatible, and their composition is $\mathcal{C} = \langle \Delta, \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}, \{\mathsf{a} \mapsto \mathsf{A}, \mathsf{b} \mapsto \mathsf{B}, \mathsf{c} \mapsto \mathsf{C}\}, \{\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}\} \rangle$ where Δ is the theory $\{\mathsf{b} \to \mathsf{a},\ \mathsf{c} \to \mathsf{b},\ (\mathsf{a} \land \mathsf{b}) \twoheadrightarrow \mathsf{c}\}$. $\mathcal{C}$ has an agreement, since $\Delta \vdash \mathsf{a} \land \mathsf{b} \land \mathsf{c}$. The agreement exploits the fact that Carl's contract allows the action c to happen "on credit", before the other actions are performed.

We now recap from [4] the notion of *proof traces*, i.e. the sequences of atoms respecting the order imposed by proofs in PCL . Consider e.g. rule ($\to$ E):

$$\frac{\Delta \vdash \alpha \to \mathsf{a} \qquad \Delta \vdash \alpha}{\Delta \vdash \mathsf{a}}\,(\to \text{E})$$

The rule requires a proof of all the atoms in α in order to construct a proof of a. Accordingly, if σ is a proof trace of Δ, then σa if a proof trace of Δ. Instead, in the rule ($\twoheadrightarrow$E), the antecedent α needs not necessarily be proved before a: it suffices to prove α by taking a as hypothesis.

Definition 17. (Proof traces [4]) *For a Horn PCL theory* Δ, *we define the set of* proof traces $[\![\Delta]\!]$ *by the rules in Fig. 5, where for* $\sigma, \eta \in E^*$ *we denote with* $\overline{\sigma}$ *the set of atoms in* σ, *with* $\sigma\eta$ *the concatenation of* σ *and* η, *and with* $\sigma \mid \eta$ *the interleavings of* σ *and* η. *We assume that both concatenation and interleaving remove duplicates from the right, e.g.* $aba \mid ca = ab \mid ca = \{abc, acb, cab\}$.

The set $\mathcal{U}_{\mathcal{C}}^{X}$ in Def. 18 contains, given a set X of atoms, the atoms which may be proved immediately after, following some proof trace of $\mathcal{C}$.

Definition 18. (Urgent actions [4]) *For a contract* $\mathcal{C} = \langle \Delta, \ldots \rangle$ *and a set of atoms* X, *we define* $\mathcal{U}_{\mathcal{C}}^{X} = \{\mathsf{a} \notin X \mid \exists \sigma, \sigma'.\ \overline{\sigma} = X \ \land\ \sigma\,\mathsf{a}\,\sigma' \in [\Delta, X]\}$.

Example 8. For the contract $\mathcal{C}$ specified by the theory $\Delta = \mathsf{a} \to \mathsf{b}, \mathsf{b} \twoheadrightarrow \mathsf{a}$, we have $[\![\Delta]\!] = \{\epsilon, \mathsf{ab}\}$, and $\mathcal{U}_{\Delta}^{\emptyset} = \{\mathsf{a}\}$, $\mathcal{U}_{\Delta}^{\{\mathsf{a}\}} = \{\mathsf{b}\}$, $\mathcal{U}_{\Delta}^{\{\mathsf{b}\}} = \{\mathsf{a}\}$, and $\mathcal{U}_{\Delta}^{\{\mathsf{a},\mathsf{b}\}} = \emptyset$.

6 From Logical to Physical Contracts

In this section we show, starting from a logical contract, how to construct a physical one which preserves the agreement property. Technically, we shall relate

$$
\begin{aligned}
S &= (\mathcal{T} \times T) \cup \big((\{\mathsf{a} \mid \bigwedge X \to \mathsf{a} \in \Delta\} \cup \{\mathsf{a} \mid \bigwedge X \twoheadrightarrow \mathsf{a} \in \Delta\} \cup \{\mathsf{a} \mid \mathsf{a} \in \Delta\}) \times \{*\}\big) \\
T &= \{(X, \mathsf{a}, \bigcirc) \mid \bigwedge X \to \mathsf{a} \in \Delta\} \cup \{(X, \mathsf{a}, \circledcirc) \mid \bigwedge X \twoheadrightarrow \mathsf{a} \in \Delta\} \\
F &= \{(s,t) \mid s = (\mathsf{a}, *),\ t = (X, \mathsf{a}, z)\} \cup \{(s,t) \mid s = (\mathsf{a}, t),\ t = (X, \mathsf{c}, z),\ \mathsf{a} \in X\} \cup \\
&\quad\ \{(t,s) \mid s = (\mathsf{a}, x),\ t = (X, \mathsf{a}, z),\ x \neq *\} \\
\Gamma(s) &= \text{if } s = (\mathsf{a}, x) \text{ with } x \in T \text{ then } \mathsf{a} \text{ else } \bot \\
\Lambda(t) &= \text{if } t = (X, \mathsf{a}, z) \text{ then } \mathsf{a} \text{ else } \bot \\
m_0(s) &= \text{if } s = (\mathsf{a}, *) \text{ then } 1 \text{ else } 0 \\
\mathcal{L} &= \{s \in S \mid s = (\mathsf{a}, t) \textit{ and } t = (X, \mathsf{c}, \circledcirc) \textit{ with } X \neq \emptyset\}
\end{aligned}
$$

Fig. 6. Translation from logical to physical contracts

provability in PCL to reachability of suitable configurations in the associated LPN. The idea of our construction is to translate each Horn clause of a PCL formula into a transition of an LPN, labelled with the action in the conclusion of the clause.

Definition 19. *Let* $\mathcal{C} = \langle \Delta, \mathcal{A}, \pi, \Omega \rangle$ *be a* PCL *contract. We define the contract net* $\mathcal{P}(\mathcal{C})$ *as* $((\langle S, T, F, \Gamma, \Lambda \rangle, m_0, \mathcal{L}), \mathcal{A}, \pi, \Omega)$ *in Fig. 6.*

The transitions associated to $\mathcal{C}$ are a subset T of $\wp(\mathcal{T}) \times \mathcal{T} \times \{\circledcirc, \bigcirc\}$. For each intuitionistic/contractual implication, we introduce a transition as follows. A clause $\bigwedge X \twoheadrightarrow \mathsf{a}$ maps to $(X, \mathsf{a}, \circledcirc) \in T$, while $\bigwedge X \to \mathsf{a}$ maps to $(X, \mathsf{a}, \bigcirc) \in T$. A formula a is dealt with as the clause $\bigwedge \emptyset \to \mathsf{a}$. Places in S carry the information on which transition may actually put/consume a token from them (even on credit). The lending places are those places (a, t) where $t = (X, \mathsf{c}, \circledcirc)$. Observe that a transition $t = (X, \mathsf{a}, z)$ puts a token in each place (a, x) with $x \neq *$, and all the transitions bearing the same labels, say a, are mutually excluding each other, as they share the unique input place $(\mathsf{a}, *)$. The initial marking will contains all the places in $\mathcal{T} \times \{*\}$, and if a token is consumed from one of these places then the place will be never marked again. Furthermore the lending places are never initially marked.

Example 9. Consider the PCL contract with formula $\mathsf{a} \twoheadrightarrow \mathsf{a}$ (the other components are immaterial for the sake of the example). The associated LPN is in

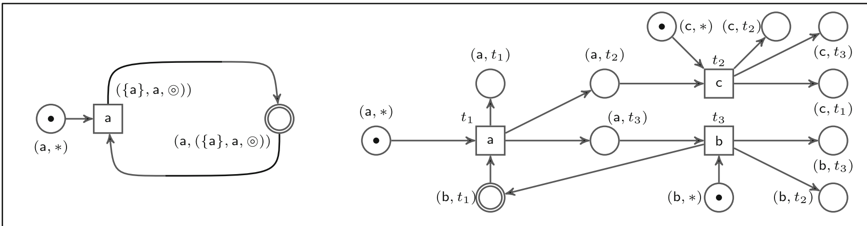


Fig. 7. Two contract nets constructed from PCL contracts

Fig. 7, left. The transition $(\{\mathsf{a}\}, \mathsf{a}, \circledcirc))$, labeled a, can be executed at the initial marking, as the unmarked place in the preset is a lending place. The reached marking contains no tokens, hence it is honored. This is coherent with the fact that $\mathsf{a} \twoheadrightarrow \mathsf{a} \vdash \mathsf{a}$ holds in PCL.

Example 10. Consider the PCL contract specified by the theory

$$\Delta \quad = \quad \{\mathsf{b} \twoheadrightarrow \mathsf{a},\ \mathsf{a} \rightarrow \mathsf{c},\ \mathsf{a} \rightarrow \mathsf{b}\}$$

The associated LPN is the one on the right depicted in Fig. 7. The transitions are $t_1 = (\{\mathsf{b}\}, \mathsf{a}, \circledcirc)$, $t_2 = (\{\mathsf{a}\}, \mathsf{c}, \bigcirc)$ and $t_3 = (\{\mathsf{a}\}, \mathsf{b}, \bigcirc)$. Initially only t_1 is enabled, lending a token from place (b, t_1). This leads to a marking where both t_2 and t_3 are enabled, but only the execution of t_3 ends up with an honored marking. The marking reached after executing all the actions is honored. This is coherent with the fact that $\Delta \vdash \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{c}$ holds in PCL.

Since all the transitions consume the token from the places $(\mathsf{a}, *)$ (where a is the label of the transition), and these places cannot be marked again, it is easy to see that each transition may occur only once. Hence, the net associated to a contract is an occurrence net. If two transitions t, t' have the same label (say a), then they cannot belong to the same state of the net. In fact, transitions with the same label share the same input place $(\mathsf{a}, *)$. This place is not a lending one, and has no ingoing arcs, hence only one of the transitions with the same label may happen. The notion of correctly labeled net lifts obviously to contract nets.

Proposition 20. *For all* PCL *contracts* $\mathcal{C}$*, the net* $\mathcal{P}(\mathcal{C})$ *is correctly labeled.*

A relevant property of $\mathcal{P}$ is that it is an homomorphism with respect to contracts composition. Thus, since both $\mid$ and $\oplus$ are associative and commutative, we can construct a physical contract from a set of logical contracts $\mathcal{C}_1 \cdots \mathcal{C}_n$ componentwise, i.e. by composing the contract nets $\mathcal{P}(\mathcal{C}_1) \cdots \mathcal{P}(\mathcal{C}_n)$.

Proposition 21. *For all* $\mathcal{C}_1, \mathcal{C}_2$*, we have that* $\mathcal{P}(\mathcal{C}_1 \mid \mathcal{C}_2) \sim \mathcal{P}(\mathcal{C}_1) \oplus \mathcal{P}(\mathcal{C}_2)$.

In Theorem 23 below we state the main result of this section, namely that our construction maps the agreement property of PCL contracts into weak termination of the associated contract nets. To prove Theorem 23, we exploit the fact that C is a set of provable atoms in the logic iff $(C, \emptyset)$ is a configuration of the associated contract net.

Lemma 22. *Let* $\mathcal{C} = \langle \Delta, \mathcal{A}, \pi, \Omega \rangle$ *be a PCL contract, and let* $\mathcal{P}(\mathcal{C}) = (O, \mathcal{A}, \pi, \Omega)$*. For all* $C \subseteq \mathcal{T}$*,* $\Delta \vdash \bigwedge C$ *iff there exists* $m \in \mathsf{M}(O)$ *such that* $\mu(m) = (C, \emptyset)$.

Theorem 23. $\mathcal{C}$ *admits an agreement iff* $\mathcal{P}(\mathcal{C})$ *weakly terminates in* Ω.

We now specialize Theorem 8, which allows for compositional verification of choreographies. Assuming a choreography specified as a PCL contract $\mathcal{C}$, we can (i) project it into the contracts $\mathcal{C}_1 \cdots \mathcal{C}_n$ of its participants, (ii) construct

the corresponding LPN contracts $\mathcal{P}(\mathcal{C}_1) \cdots \mathcal{P}(\mathcal{C}_n)$, and (*iii*) individually refine each of them into a service implementation. If the original choreography admits an agreement, then the composition of the services weakly terminates, i.e. it is correct w.r.t. the choreography.

Theorem 24. *Let* $\mathcal{C} = \mathcal{C}_1 \mid \cdots \mid \mathcal{C}_n$ *admit an agreement, with* Ω_i *goals of* $\mathcal{C}_i$. *If* $\mathcal{D}_i$ *refines* $\mathcal{P}(\mathcal{C}_i)$ *for* $i \in 1..n$, *then* $\mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$ *weakly terminates in* $\Omega_1 \cup \cdots \cup \Omega_n$.

The notion of urgency in contract nets correspond to that in the associated PCL contracts (Theorem 25).

Theorem 25. *For all* PCL *contracts* $\mathcal{C}$, *and for all* $X \subseteq \mathcal{T}$, $\mathcal{U}^X_{\mathcal{C}} = \mathcal{U}^X_{\mathcal{P}(\mathcal{C})}$.

Example 11. Recall from Ex. 8 that, for $\mathcal{C} = \langle\{\mathsf{a} \rightarrow \mathsf{b}, \mathsf{b} \twoheadrightarrow \mathsf{a}\}, \ldots\rangle$, we have:

$$\mathcal{U}^{\emptyset}_{\mathcal{C}} = \{\mathsf{a}\} \qquad \mathcal{U}^{\{\mathsf{a}\}}_{\mathcal{C}} = \{\mathsf{b}\} \qquad \mathcal{U}^{\{\mathsf{b}\}}_{\mathcal{C}} = \{\mathsf{a}\} \qquad \mathcal{U}^{\{\mathsf{a},\mathsf{b}\}}_{\mathcal{C}} = \emptyset$$

This is coherent with the fact that, in the corresponding contract net $N'' \oplus N'$ in Fig. 3, only a is urgent at the initial marking, while b becomes urgent after a has been fired.

7 Related Work and Conclusions

We have investigated how to compile logical into physical contracts. The source of the compilation is the Horn fragment of Propositional Contract Logic [7], while the target is a contract model based on lending Petri nets (LPNs). Our compilation preserves agreements (Theorem 23), as well as the possibility of protecting services against misbehavior of malevolent services. LPN contracts can be used to reason compositionally about the realization of a choreography (Theorem 24), so extending a result of [21]. Furthermore, we have given a logical characterization of those *urgent* actions which have to be performed in a given state. This notion, which was only intuitively outlined in [7], is now made formal through our compilation into LPNs (Theorem 25).

Contract nets seem a promising model for reasoning on contracts: while having a clear relation with PCL contracts, they may inherit as well the whole realm of tools that are already available for Petri nets.

The notion of places with a negative marking is not a new one in the Petri nets community, though very few papers tackle this notion, as the interpretation of *negative* tokens does not match the intuition of Petri nets, where tokens are generally intended as resources. In this paper we have used negative tokens to model situations where actions are in a *circular* dependency, like the ones arising in PCL contracts. Lending places model the intuition that an action can be performed on a *promise*, and a negative token in a place can be interpreted as the promise made, which must be, sooner or later, *honored*. Indeed, the net obtained from a PCL contract is an occurrence net which may contain cycles,

e.g. in the net of Ex. 10 the transition t_1 depends on t_3, which in turn depends on t_1 (and to execute t_1 we required to *lend* a token which is after supplied by t_3). In [20] the idea of places with negative marking is realized using a new kind of arc, called *debit* arcs. Under suitable conditions, these nets are Turing powerful, whereas our contract nets do not add expressiveness (while for LPNs the issue has to be investigated). In [17] negative tokens arise as the result of certain *linear* assumptions. The relations with LPNs have to be investigated.

Acknowledgements. We thank Philippe Darondeau, Eric Fabre and Roberto Zunino for useful discussions and suggestions. This work has been partially supported by Aut. Reg. of Sardinia grants L.R.7/2007 CRP2-120 (TESLA) CRP-17285 (TRICS) and P.I.A. 2010 ("Social Glue"), and by MIUR PRIN 2010-11 project "Security Horizons".

References

1. Abadi, M., Burrows, M., Lampson, B., Plotkin, G.: A calculus for access control in distributed systems. ACM TOPLAS 4(15) (1993)
2. Abadi, M., Plotkin, G.D.: A logical view of composition. TCS, 114(1) (1993)
3. Armbrust, M., et al.: A view of cloud computing. Comm. ACM 53(4), 50–58 (2010)
4. Bartoletti, M., Cimoli, T., Di Giamberardino, P., Zunino, R.: Contract agreements via logic. In: Proc. ICE (2013)
5. Bartoletti, M., Cimoli, T., Pinna, G.M., Zunino, R.: An event-based model for contracts. In: Proc. PLACES (2012)
6. Bartoletti, M., Cimoli, T., Zunino, R.: A theory of agreements and protection. In: Basin, D., Mitchell, J.C. (eds.) POST 2013. LNCS, vol. 7796, pp. 186–205. Springer, Heidelberg (2013)
7. Bartoletti, M., Zunino, R.: A calculus of contracting processes. In: LICS (2010)
8. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010)
9. Bravetti, M., Lanese, I., Zavattaro, G.: Contract-driven implementation of choreographies. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 1–18. Springer, Heidelberg (2009)
10. Bravetti, M., Zavattaro, G.: Contract based multi-party service composition. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 207–222. Springer, Heidelberg (2007)
11. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
12. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Transactions on Programming Languages and Systems 31(5) (2009)
13. Even, S., Yacobi, Y.: Relations among public key signature systems. Technical Report 175, Computer Science Department, Technion, Haifa (1980)
14. Garg, D., Abadi, M.: A modal deconstruction of access control logics. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 216–230. Springer, Heidelberg (2008)

15. Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: Proc. PLACES (2010)
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL (2008)
17. Martí-Oliet, N., Meseguer, J.: An algebraic axiomatization of linear logic models. In: Topology and Category Theory in Computer Science (1991)
18. Prisacariu, C., Schneider, G.: A dynamic deontic logic for complex contracts. The Journal of Logic and Algebraic Programming (JLAP) 81(4) (2012)
19. Reisig, W.: Petri Nets: An Introduction. Monographs in Theoretical Computer Science. An EATCS Series, vol. 4. Springer (1985)
20. Stotts, P.D., Godfrey, P.: Place/transition nets with debit arcs. Inf. Proc. Lett. 41(1) (1992)
21. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: Agreeing and implementing interorganizational processes. Comput. J. 53(1) (2010)
22. van Glabbeek, R.J., Plotkin, G.D.: Configuration structures. In: LICS (1995)

On Efficiency Preorders*

Manish Gaur[1,2] and S. Arun-Kumar[3]

[1] School of Computing, University of Glasgow, Glasgow, Scotland
[2] Department of Computer Science and Engineering,
Institute of Engineering and Technology, Lucknow
[3] Department of Computer Science and Engineering,
Indian Institute of Technology Delhi, New Delhi, India

Abstract. Theories of efficiency preorders and precongruences for concurrent systems have been described in various papers. We describe a procedure to implement two of these precongruences. Considering the extra information that is needed to be maintained while computing efficiency preorders, our procedure with a complexity $O(n^3m)$, compares favourably with that for deciding observational equivalence ($O(n^{\alpha}m)$). Further, the algorithm may be plugged in to existing model-checkers such as the Concurrency-Workbench of the New Century (CWB-NC) without any significant overheads of space or time.

1 Introduction

Research in process algebra has focused on the use of behavioural relations such as equivalences and refinement orderings as a basis for establishing system correctness. In the process algebraic framework, both specifications and implementations are defined in the same language; the intuition is that a specification describes the desired high level behaviour, while the implementation details the proposed means for achieving this behaviour. One then uses an appropriate equivalence or preorder to establish that an implementation behaves as defined in the specification. In the case of equivalence based reasoning, an implementation is correct if its behaviour is indistinguishable from that of its specification. Refinement (or Preorder) relations, on the other hand, typically embody a notion of comparison: an implementation conforms to (or refines) a specification if the behaviour of the former is "at least as good as" that stipulated by the specification. The benefits of such process algebraic approaches include the following:

- Users as well as testing and verification tools work within a single formalism for specification and implementation.
- The algebra provides explicit support for *compositional* specification and implementation, allowing the specification (implementation) of a system to be built up from the specification (implementation) of its components.
- Specifications include information about what is disallowed as well as what is allowed.

* With support from Sun Microsystems Inc. USA

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 83–94, 2013.
DOI: 10.1007/978-3-642-40213-5_6,
© IFIP International Federation for Information Processing 2013

Consequently, a number of different process algebras have been studied, and a variety of different equivalences and refinement relations capturing different aspects of behaviour have been developed. The simplest of these equivalences is the notion of bisimulation and there exist efficient algorithms and tools to implement various bisimulation-based preorders and equivalences. In general, algorithms for computing semantic equivalences and preorders usually consist of two steps. In the first step, the entire state space is generated and the second step then manipulates this space to determine whether an appropriate relation exists. Such algorithms are usually referred to as "*global*", as opposed to "*on-the-fly*" or "*local*" algorithms which work on a partially generated state space in an attempt to mitigate the state explosion problem in verification.

Refinements could come in several flavours. One of the earliest refinement relations ([10], [11]) in the process algebra framework related refinement to the level of indeterminacy in the specification i.e. a more determinate process was considered a refinement of a less determinate one in terms of behaviour. Other notions of refinement such as action refinement yield various other preorders.

One such method is efficiency prebisimulation for processes ([4], [2], [3]). It is based on the simple idea of, essentially, counting the number of internal moves by a process. It has also been shown that it can be incorporated within the general frame work of bisimulation, to obtain a mathematically tractable preorder, which in common with the standard notions of bisimulation equivalence, is sensitive to the branching structure of processes.

Hence in the context of verification and development methodology it is more fruitful to regard these preorders as particular refinement relations. Due to the state explosion problem in verifying concurrent systems, it makes more sense to adopt a top-down methodology in both the development and verification of concurrent systems.

An alternative method to alleviate the state explosion problem is to use congruences and precongruences on the specification language as detailed below.

Consider a specification S^0 of a system. An initial refinement of this system would typically split the specification into (specifications of) subsystems

$$S_1, S_2, \ldots, S_n$$

combined in some fashion to obtain a first refinement

$$S^1 = o(S_1, S_2, \ldots, S_n)$$

where o is some appropriately defined combinator. Under a typical precongruence $\leq^c$ as defined in ([4], [2], [3]) we would have $S^1 \leq^c S^0$. The problem now reduces to obtaining refinements $(S_1^1, S_2^1, \ldots, S_n^1)$ such that for each $i, 1 \leq i \leq n, S_i^1 \leq^c S_i$. It is then clear from the properties of $\leq^c$ that

$$S^2 = o(S_1^1, \ldots, S_n^1) \leq^c o(S_1, \ldots, S_n) \leq^c S^0$$

Hence the problem of developing and verifying a large system to satisfy a specification S may be broken down to the problem of performing n smaller verification problems viz. $S_i^1 \leq^c S_i (1 \leq i \leq n)$.

The above verification methodology has the following advantages:

1. It closely follows the development methodology rather than postponing the problem of verification of a possible complex system to the end.
2. This methodology allows for the verification of the large system to be directly inferred from the verification of the smaller subsystems (without actually performing the verification).

A characteristic of process-oriented behavioural relations is that they are usually defined on *labeled transition systems*, which forms the semantic model of systems, rather than with respect to a particular syntax of process descriptions. This style of definition permits notions of equivalence and refinement to be applied to any algebra with a semantics given in terms of labeled transition systems. If in addition, these labeled transition systems are finite state, then the relation may be calculated in a purely mechanical manner: algorithms may then be developed for automatically checking that an implementation satisfies a specification.

In this paper we propose a method for computing efficiency prebisimulations for processes which are in fact bisimulation-based refinement relations called efficiency preorders. There exist very good algorithms [17,13,9] and tools such as [18] to construct strong and weak bisimulations on labeled transition systems. But little attention is given in computing the efficiency preorders. We believe that incorporating the feature of computation of efficiency preorder on a labeled transition system will add significant power to the verification tools like CWB-NC [18].

Theories of efficiency preorders (which combine both correctness and efficiency considerations into a single preorder) have been developed in various papers ([2], [4], [3]). The largest precongruences contained in these preorders have also been characterized and axiomatized for finite CCS [15] processes. The notion of efficiency in these cases is abstract enough to be interpreted loosely as based on timing, communication or even energy consumed in a computation. Efficiency Preorder has been studied in modeling various distributed systems, where a notion of comparison is used reflect the fact one implementation is at least as good as another. Most recently the concept of efficiency preorder has been used in [8] in context of an extension of asynchronous pi-calculus[16].

Many equivalence and preorder checking problems on labelled transition systems may be reduced to the problem of constructing a strong bisimulation. In any labeled transition system of finite state processes if n is the total number of states and m the number of transitions then Paige and Tarjan [17] gave an $O(m\log(n))$ solution to the generalised partitioning problem on some relation E on states of FSPs. Kanellakis and Smolka studied the problem of equivalence checking of CCS expression and gave $O(m\log(n)+n)$ and $O(n^2m\log(n)+mn^{\alpha})$ time algorithms for strong bisimulation and weak bisimulation respectively [13]. Here the smallest such α known is 2.376 [7]. We propose a method running in $O(mn^3)$ time complexity for deciding efficiency preorders.

The organization of the paper is as follows: In section 2 we give the basic definitions of labeled transition systems and various equivalences and preorders

relevant to our purpose and characterize them. In Section 3 we describe a method for constructing efficiency preorder. It is a polynomial time algorithm whose complexity is no more than that of deciding weak bisimulation. Section 4 is the conclusion. It briefly compares the time complexty of our method with that of deciding weak bisimulation.

2 Basic Definitions and Characterization

In this section we will define and characterize a general framework in which the labelled transition system (LTS) on processes will be used to present an algorithm for deciding efficiency prebisimulations.

Definition 1. A *Finite State Process (FSP)* is a 5-tuple

$$\langle K, p_0, A, \longrightarrow, X \rangle$$

where

- K is a finite set of *states*,
- $p_0 \in K$ is the *start* state,
- A is a finite set of labels,
- $\longrightarrow \subseteq K \times A \times K$ is the *transition*[1] relation,
- $X \subseteq K \times \{x\}$, where $x \notin A$, is the *extension* relation[2].

LTSs tell us what behaviour of process is. The next question now is: when should two behaviours be considered equal? That is, what does it mean that two processes are equivalent? Intuitively, two processes should be equivalent if they cannot be distinguished by interacting with them. It's easy to observe that LTSs resemble graphs and the standard equality on graphs is *graph isomorphism*. In [15,20] it has been shown that graph isomorphism is too strong as a behavioural equivalences for processes. It prevents us equating processes that should be considered equal.

Another important notion of equivalence is present by the *automata theory* in computer science. The notion of automata and LTS are very similar and two automata are considered equal if they accept same set of labels (actions)[12]. The analogous equivalence on processes is called *trace equivalence*. But the trace equivalence as behaviour equality for processes is also not acceptable [15]. In fact for process equivalence we need a loose equality than graph isomorphism but a tighter correspondence between transitions than trace equivalence. Intuitively when some transition is done by a process the other must be able to mimic it and the evolved process out of these transitions must in turn be able to do the same again. This idea of equality, known as *bisimilarity* has been extensively studied[15,5] in process algebra. We will formally define this in the following definition.

[1] we write $p \longrightarrow^{\alpha} q$ to denote $\langle p, \alpha, q \rangle \in \longrightarrow$.

[2] The extension relation has been kept for completeness and will not be used in the paper except to be referred in the conclusion.

Definition 2. Let $P = \langle K, p_0, A, \longrightarrow, X\rangle$ be a FSP and let ρ and σ be binary relations on Σ. A binary relation R on K is a (ρ,σ)*-induced bisimulation* if pRq implies the following conditions hold, for all $\alpha, \beta \in \Sigma$.

1. $p \longrightarrow^{\alpha} p' \Rightarrow \exists\beta, q' : \alpha\rho\beta \wedge q \longrightarrow^{\beta} q' \wedge p'Rq'$, and
2. $q \longrightarrow^{\beta} q' \Rightarrow \exists\alpha, p' : \alpha\sigma\beta \wedge p \longrightarrow^{\alpha} p' \wedge p'Rq'$.

A $(=,=)$-induced bisimulation will sometimes be called a *natural bisimulation* on a finite-state process.

A FSP may be represented by a labeled directed graph whose nodes are states and whose arcs have labels from A. We will be particularly interested in the case where the the set of labels of a FSP is a finite set A of symbols called *actions* such that $\tau \in A$ is a distinguished action called the *invisible* action and $V = A - \{\tau\}$ is the set of *visible* actions.

Let A^* denote the set of all finite sequences of actions (including the empty sequence ε). We write $\hat{s}$ to denote the sequence obtained from $s \in A^*$ by deleting all occurrences of τ. If s contains no visible action then $\hat{s}$ yields ε. Finally $|s|$ denotes the length of the sequence s. We write $s \hat{=} t$ if $\hat{s} = \hat{t}$.

For $s, t \in A^*$ and $a \in A$, the transitions $p \longrightarrow^{s} p'$ and $p \Longrightarrow^{s} p'$ are defined by induction on the length of s as follows

- $p \longrightarrow^{\varepsilon} p$ for all p,
- $p \longrightarrow^{s} p'$ for $s = ta$ iff $\exists p'' : p \longrightarrow^{t} p'' \longrightarrow^{a} p'$,
- $p \Longrightarrow^{\varepsilon} p'$ iff $\exists m \geq 0 : p \longrightarrow^{\tau^m} p'$,
- $p \Longrightarrow^{a} p'$ iff $\exists p'', p''' : p \Longrightarrow^{\varepsilon} p'' \longrightarrow^{a} p''' \Longrightarrow^{\varepsilon} p'$, and
- $p \Longrightarrow^{s} p'$ for $s = ta$ iff $\exists p'' : p \Longrightarrow^{t} p'' \Longrightarrow^{a} p'$.

Let $\preceq$ be the relation on A^* generated by the inequations $s \preceq s$ and $\tau s \preceq s$, i.e. $\preceq$ is closed under reflexivity, transitivity and substitution under catenation contexts. It is clear that $\varepsilon \preceq \tau$, $s\tau \preceq s$ for all s and that $\preceq$ is antisymmetric. Hence $\preceq$ is a partial order on A^* and $s = t$ iff $s \preceq t$ and $t \preceq s$. We will be particularly interested in the set of *extended* actions defined by $EA = \{u \in A^* \mid |\hat{u}| \leq 1\}$, viz. the set of sequences which contain at most one visible action. It is easy to see that for any $a \in V$ and $v \in EA$, $a \preceq v$ implies $v = a$. Also $\tau^i \preceq \tau^j$ iff $i \geq j$.

Bisimulations can be regarded as one of the most important contributions of concurrency theory to computer science. Nowadays, bisimulation and co-inductive techniques [20] developed from the idea of bisimulation are widely used. Here we define two earliest discovered bisimulations called as strong and weak bisimualtion.

Definition 3. A binary relation R on the states of a FSP $\langle K, p_0, A, \longrightarrow, X\rangle$, is

- a *strong bisimulation* (also $\sim$*-bisimulation*) if pRq implies for every $a \in A$,
  1. $p \longrightarrow^{a} p' \Rightarrow \exists q' : q \longrightarrow^{a} q' \wedge p'Rq'$, and
  2. $q \longrightarrow^{a} q' \Rightarrow \exists p' : p \longrightarrow^{a} p' \wedge p'Rq'$.

– a *weak bisimulation* (also $\approx$-*bisimulation*) if pRq implies for every $a \in A$
  1. $p \longrightarrow^{a} p' \Rightarrow \exists q' : q \Longrightarrow^{\hat{a}} q' \wedge p'Rq'$ and
  2. $q \longrightarrow^{a} q' \Rightarrow \exists p' : p \Longrightarrow^{\hat{a}} p' \wedge p'Rq'$.

Some enhancements were proposed shortly after discovery of strong and weak bisimulation. The best known example is Milner's *bisimulation up to bisimilarity* technique [14], in which the closure of the bisimulation relation is achieved up to bisimilarity itself. The up to bisimilarity is basically achieved with the fact that $\sim$ is a transitive relation. A problem with up to bismilarity is that the it fails for weak bisimulation despite it being transitive [19,21]. In its place, a number of variations have been proposed; for instance allowing only uses of strong bisimilarity with in the upto bisimilarity [14]. The most important variation, however, involves a relation called *expansion* [3,21]. Expansion is a preorder derived from weak bismilarity by, essentially, comparing the number of silent actions. The idea underlying expansion is roughly that if Q expands P, then P and Q are bisimilar, except that in mimicking P's behaviour, Q cannot perform more τ transitions than P. We can think of P uses at least as many resources as Q. An interest of expansion derives from the fact that, in practice, most of weak bisimilarity are indeed instances of expansion. Expansion is preserved by all CCS [15] operators but sum, and has a complete proof system for finite terms based of a modification of the standard τ laws for CCS. Expansion is also a powerful auxiliary relation for up to techniques involving weak forms of behaviour equivalences. Now we define following two expansions called *efficiency prebisimulation* and *elaboration*. These expansions typically embody a notion of efficiency where one process is at least as efficient as the other provided they are behaviourally equivalent.

Definition 4. A binary relation R on the states of a FSP $\langle K, p_0, A, \longrightarrow, X\rangle$, is

– an *efficiency prebisimulation* (also $\lesssim$-*bisimulation*) if pRq implies for every $u, v \in EA$,
  1. $p \longrightarrow^{u} p' \Rightarrow \exists v, q' : u \preceq v \wedge q \longrightarrow^{v} q' \wedge p'Rq'$, and
  2. $q \longrightarrow^{v} q' \Rightarrow \exists u, p' : u \preceq v \wedge p \longrightarrow^{u} p' \wedge p'Rq'$.

– an *elaboration* (also $\lessapprox$-*bisimulation*) if pRq implies for every $a \in A$,
  1. $p \longrightarrow^{a} p' \Rightarrow \exists q' : q \Longrightarrow^{\hat{a}} q' \wedge p'Rq'$, and
  2. $q \longrightarrow^{a} q' \Rightarrow \exists p' : p \Longrightarrow^{a} p' \wedge p'Rq'$.

Definitions 3 and 4 are from [2,4]. The following proposition is a direct consequence of these definitions. In Proposition 5, strong bisimulation, weak bisimulation, efficiency prebisimulation and elaboration are expressed over a sequence of actions.

Proposition 5. *A binary relation R on the states of a FSP, is*

– *a strong bisimulation if pRq implies for every $s \in A^*$,*
  1. *$p \longrightarrow^{s} p'$ then $\exists q' : q \longrightarrow^{s} q' \wedge p'Rq'$ and*
  2. *$q \longrightarrow^{s} q'$ then $\exists p' : p \longrightarrow^{s} p' \wedge p'Rq'$.*

– *a weak bisimulation if pRq implies for every* $s, t \in A^*$
  1. $p \longrightarrow^s p' \Rightarrow \exists q', t : \hat{s} = \hat{t} \wedge q \longrightarrow^t q' \wedge p'Rq'$, *and*
  2. $q \longrightarrow^t q' \Rightarrow \exists p', s : \hat{s} = \hat{t} \wedge p \longrightarrow^s p' \wedge p'Rq'$.
– *an elaboration if pRq implies for every* $s, t \in A^*$,
  1. $p \longrightarrow^s p' \Rightarrow \exists q', t : \hat{s} = \hat{t} \wedge q \longrightarrow^t q' \wedge p'Rq'$,
  2. $q \longrightarrow^t q' \Rightarrow \exists p', s : \hat{s} = \hat{t}, |s| \geq |t| \wedge p \longrightarrow^s p' \wedge p'Rq'$.
– *an efficiency prebisimulation if pRq implies for every* $s, t \in A^*$,
  1. $p \longrightarrow^s p' \Rightarrow \exists q', t : \hat{s} = \hat{t}, |s| \geq |t| \wedge q \longrightarrow^t q' \wedge p'Rq'$,
  2. $q \longrightarrow^t q' \Rightarrow \exists p', s : \hat{s} = \hat{t}, |t| \leq |s| \wedge p \longrightarrow^s p' \wedge p'Rq'$.

Proposition 6. *The following facts are then easily proven [2,4].*

1. *Every strong bisimulation is an efficiency prebisimulation.*
2. *Every efficiency prebisimulation is an elaboration.*
3. *Every elaboration is a weak bisimulation.*
4. *For* $\sqsubseteq \in \{\sim, \lesssim, \lessapprox, \approx\}$, *the largest* $\sqsubseteq$*-bisimulation, denoted* $\sqsubseteq$, *is a preorder.*
5. *For* $\sqsubseteq \in \{\sim, \lesssim, \lessapprox, \approx\}$, $p \sqsubseteq q$ *iff there exists a* $\sqsubseteq$*-bisimulation containing* (p, q).
6. *The largest* $\sim$*- and* $\approx$*-bisimulations, are equivalence relations.*

The following simple examples in CCS syntax [15] give some idea of the distinctions between the relations discussed above. For CCS operators none of the relations $\lesssim, \lessapprox, \approx$, preserves summation. It is therefore necessary to consider the largest (pre)congruence contained in $\sqsubseteq$ (and denoted $\sqsubseteq^c$) in order to be able to use refinement effectively.

Example 7. Let a be a visible action. Then

1. $a.\tau.\mathbf{0} \lessapprox^c a.\mathbf{0}$ *but the converse does not hold.*
2. $a.\mathbf{0} + a.\tau.\mathbf{0} \lesssim^c a.\mathbf{0}$ *but the converse does not hold.*
3. $a.\mathbf{0} + a.\tau.\tau.\mathbf{0} \lessapprox^c a.\tau.\mathbf{0}$ *but* $a.\mathbf{0} + a.\tau.\tau.\mathbf{0} \not\lesssim^c a.\tau.\mathbf{0}$

For $s, t \in A^*$, let $s \preceq\cdot\, t$ if $\hat{s} = \hat{t}$ and $|s| \geq |t|$ and $s \doteq t$ if $s \preceq\cdot\, t$ and $t \preceq\cdot\, s$. Clearly $\widehat{=}$ is a strictly coarser relation than $\doteq$. Also for any $a \in V$ and $v \in EA$, $a \preceq\cdot\, v$ implies $v = a$, and $\tau^i \preceq\cdot\, \tau^j$ iff $i \geq j$. Further, $\preceq\cdot$ is coarser than $\preceq$ (i.e. $u \preceq v$ implies $u \preceq\cdot\, v$ but not the converse). We are now ready with the following lemma which is used in the characterization of Theorem 10.

Lemma 8. *Let R be a binary relation on the states of a FSP and pRq. The following are equivalent.*

1. *For all* $a \in A$, $p \longrightarrow^a p' \Rightarrow \exists q' : q \Longrightarrow^a q' \wedge p'Rq'$.
2. *For all* $u \in EA$, $p \longrightarrow^u p' \Rightarrow \exists v \in EA, q' : u \preceq v \wedge q \longrightarrow^v q' \wedge p'Rq'$.
3. *For all* $u \in EA$, $p \longrightarrow^u p' \Rightarrow \exists v \in EA, q' : u \preceq\cdot\, v \wedge q \longrightarrow^v q' \wedge p'Rq'$.
4. *For all* $s \in A^*$, $p \longrightarrow^s p' \Rightarrow \exists t \in A^*, q' : s \preceq t \wedge q \longrightarrow^t q' \wedge p'Rq'$.
5. *For all* $s \in A^*$, $p \longrightarrow^s p' \Rightarrow \exists t \in A^*, q' : s \preceq\cdot t \wedge q \Longrightarrow^t q' \wedge p'Rq'$.

Proof. In [2] it has been shown that (1) is equivalent to (2). It is also clear that (2) implies (3) since $u \preceq v$ implies $u \preceq\!\cdot\, v$.

It is easy to see that (2),(3), (4) and (5) all imply (1) since $\hat{a} = a$ for $a \in V$ and $\hat{a} = \varepsilon$ if $a = \tau$. Similarly it is easy to see that (4) implies (2) and (5) implies (3) by restricting (4) and (5) respectively to extended actions.

By similar reasoning, (4) implies (5). That (2) implies (4) and (3) implies (5) may be easily shown by splitting up the transition $p \longrightarrow^s p'$ into a sequence of transitions over extended actions.

($3 \Rightarrow 2$). Assume $p \longrightarrow^u p'$. If $\hat{u} = \varepsilon$, then $u = \tau^i$ for some $i \geq 0$. It follows that for some $m \geq i$, $q \longrightarrow^{\tau^m} q' \wedge p'Rq'$ and the case is proved. On the other hand if $\hat{u} = a \in V$, then $u = \tau^i a \tau^j$ for some $i, j \geq 0$. Hence there exist p_i, p_j, such that $p \longrightarrow^{\tau^i} p_i \longrightarrow^a p_j \longrightarrow^{\tau^j} p'$. By the conditions of (3) it follows that there exist $m \geq i$, $n \geq j$ and states q_m, q_n and q' such that $q \longrightarrow^{\tau^m} q_m \longrightarrow^a q_n \longrightarrow^{\tau^n} q'$ and $p_i R q_m$, $p_j R q_n$ and $p'Rq'$. Clearly therefore for $v = \tau^m a \tau^n$, q', we have that (2) holds. □

From Definitions 2, 3, 4 and Lemma 8 it is easy to see the following corollary

Corollary 9. *In any graph representing a FSP,*

1. *a strong bisimulation is a natural bisimulation,*
2. *an efficiency prebisimulation is a* $(\preceq, \preceq)$*-induced bisimulation,*
3. *an elaboration is a* $(\widehat{=}, \preceq\!\cdot)$*-induced bisimulation, and*
4. *a weak bisimulation is a* $(\widehat{=}, \widehat{=})$*-induced bisimulation.*

We then have the following characterization of the two prebisimulations which will be used to present the algorithm to decide them in next section.

Theorem 10. *(Characterization).*

- *The following are equivalent for any binary relation R on the states of a FSP.*
  1. *R is an efficiency prebisimulation.*
  2. *pRq implies for all $u, v \in EA$,*
     $p \longrightarrow^u p' \Rightarrow \exists v, q : u \preceq\!\cdot\, v \wedge q \longrightarrow^v q' \wedge p'Rq'$ *and*
     $q \longrightarrow^v q' \Rightarrow \exists u, p' : u \preceq\!\cdot\, v \wedge p \longrightarrow^u p' \wedge p'Rq'$.
  3. *pRq implies for all $s, t \in A^*$,*
     $p \longrightarrow^s p' \Rightarrow \exists q', t : s \preceq\!\cdot\, t \wedge q \longrightarrow^t q' \wedge p'Rq'$ *and*
     $q \longrightarrow^t q' \Rightarrow \exists p', s : s \preceq\!\cdot\, t \wedge p \longrightarrow^s p' \wedge p'Rq'$.
- *and so are the following.*
  1. *R is an elaboration.*
  2. *pRq implies for all $u, v \in EA$,*
     $p \longrightarrow^u p' \Rightarrow \exists v, q : u \widehat{=} v \wedge q \longrightarrow^v q' \wedge p'Rq'$ *and*
     $q \longrightarrow^v q' \Rightarrow \exists u, p' : u \preceq\!\cdot\, v \wedge p \longrightarrow^u p' \wedge p'Rq'$.
  3. *pRq implies for all $s, t \in A^*$,*
     $p \longrightarrow^s p' \Rightarrow \exists q', t : s \widehat{=} t \wedge q \longrightarrow^t q' \wedge p'Rq'$ *and*
     $q \longrightarrow^t q' \Rightarrow \exists p', s : s \preceq\!\cdot\, t \wedge p \longrightarrow^s p' \wedge p'Rq'$.

The above characterization shows that the nature of efficiency prebisimulations remains unchanged even when the preorder $\preceq$ is weakened to $\preceq\cdot$. This fact provides us a convenient handle on which to base our algorithm.

Corollary 11. *A binary relation R on the states of a FSP $\langle K, p_0, EA, \longrightarrow, X\rangle$, is*

- *an* efficiency prebisimulation *iff it is a $(\preceq\cdot, \preceq\cdot)$-induced bisimulation*
- *an* elaboration *iff it is a $(\widehat{=}, \preceq\cdot)$-induced bisimulation.*

3 The Algorithm

For finite state processes with n the number of states and m the number of transitions Paige and Tarjan [17] gave an $O(mlog_2(n))$ solution to a generalised partitioning problem. Kanellakis and Smolka studied the problem of checking equivalences of CCS expressions and gave $O(mlog_2(n)+n)$ and $O(n^2 mlog_2(n)+mn^\alpha)$ (where $2 < \alpha \leq 3$) algorithms for strong and weak bisimulation respectively. In this section we present a method for computing prebisimulations.

A direct consequence of Theorem 10 is that the extended actions $\tau^i a \tau^j$ and $\tau^m a \tau^n$ are indistinguishable whenever $i+j = m+n$. It suffices therefore to consider the set $EA' = \{(V \cup \{\varepsilon\}) \times \mathbb{N}\}$ (where $\mathbb{N}$ is the set of naturals) as representing the set of extended actions. For any $\langle a, m\rangle \in EA'$, we define $p \Longrightarrow^a_m p'$ to mean $\exists i, j : i+j = m \wedge p \longrightarrow^{\tau^i a \tau^j} p'$ and reserve the notation $p \Longrightarrow^a p'$ to mean $\exists m : p \Longrightarrow^a_m p'$.

Consider the FSP $P = \langle K, p_0, A, \longrightarrow, X\rangle$ and the underlying directed graph $G = \langle K, \longrightarrow^\tau\rangle$ represented by a function $\lambda : K \times K \longrightarrow (\mathbb{N} \cup \{\infty\})$ defined as

$$\lambda(p,q) = \begin{cases} 0 & \text{if} i = j \\ 1 & \text{if} i \longrightarrow^\tau j \\ \infty & \text{otherwise} \end{cases}$$

For any path $\pi = (p_1, \ldots, p_k)$ in G we define the length of the path π as $len(\pi) = \sum_{j=1}^{k-1} \lambda(p_j, p_{j+1})$. Let $Q^k_{i,j}$ be the set of paths from vertex p_i to p_j with all intermediate vertices in the set $\{p_1, \ldots, p_k\}$. Let $len^k_{i,j} = min_{\pi \in Q^k_{i,j}} len(\pi)$. It follows that

$$len^k_{i,j} = min(len^{k-1}_{i,j},\ len^{k-1}_{i,k} + len^{k-1}_{k,j}) \tag{1}$$

We may use dynamic programming [22,1] to solve the recurrence (1) for all values of i, j and k. If λ is represented by an $n \times n$ adjacency matrix, then the solution to recurrence (1) yields a $n \times n$-matrix M_{τ^*}, where $|K| = n$.

For each $\alpha \in V \cup \{\varepsilon\}$, let M_α be an $n \times n$ matrix of ordered pairs, whose first component is a boolean value and the second component is a natural number. Then we have

$$M_\varepsilon(i,j) = \begin{cases} (0,0) & \text{if} M_{\tau^*}(i,j) = \infty \\ (1, M_{\tau^*}(i,j)) & \text{otherwise} \end{cases}$$

and for each $a \in V$,

$$M_a(i,j) = \begin{cases} (1,0) \text{ if } i \longrightarrow^a j \\ (0,0) \text{ otherwise} \end{cases}$$

For each $a \in V$ we may then compute the matrix Δ_a as follows. $\Delta_a(i,l) = M_\varepsilon(i,j).M_a(j,k).M_\varepsilon(k,l)$, where $(b,x).(c,y) = (b \wedge c, x+y)$. Let Δ_a^* be the matrix containing only the first components of Δ_a.

It is easy to see that $p_i \Longrightarrow_m^a p_k$ iff $\Delta_a(i,l) = (1,m)$. Let $\Delta^\dagger = \bigcup_{a \in V \cup \{\varepsilon\}} \Delta_a$ and $\Delta^* = \bigcup_{a \in V \cup \{\varepsilon\}} \Delta_a^*$. Given an FSP $P = \langle K, p_0, A, \longrightarrow, X\rangle$, we may therefore construct the FSP $P^\dagger = \langle K, p_0, EA', \Delta^\dagger, X\rangle$ by the above procedure. By simply ignoring the second component in each element of the matrix $\Delta^\dagger$ we obtain also the FSP $P^* = \langle K, p_0, V \cup \{\varepsilon\}, \Delta^*, X\rangle$. For $\alpha = \langle a, m\rangle, \beta = \langle b, n\rangle \in \Delta^\dagger$, let $\alpha \leq\cdot \beta$ if and only if $a = b$ and $m \leq n$. Further let $\alpha \widehat{=} \beta$ if and only if $a = b$. Then

Proposition 12. *Let $P = \langle K, p_0, A, \longrightarrow, X\rangle$ be a FSP and let $P^\dagger$ and P^* be the FSPs obtained by from P by the above procedure. Then for any states $p, q \in K$,*

1. *$p \sim q$ in P iff there exists a natural bisimulation R on the states of $P^\dagger$ with pRq.*
2. *$p \lesssim q$ in P iff there exists a $(\leq\cdot, \leq\cdot)$-induced bisimulation R on the states of $P^\dagger$ with pRq.*
3. *$p \lessapprox q$ in P iff there exists a $(\widehat{=}, \leq\cdot)$-induced bisimulation R on the states of $P^\dagger$ with pRq.*
4. *$p \approx q$ in P iff there exists a natural bisimulation R on the states of P^* with pRq.*

Proof. Directly follows from the construction of the transition relation $\Delta^\dagger$. Note that comparison under $\preceq\cdot$ involves comparing second components, whenever the transitions exist under $\Delta^\dagger$. By ignoring the second component in each element of $\Delta^\dagger$, we may compare two elements in $\Delta^\dagger$ under $\widehat{=}$. □

Theorem 13. *Let p, q be states of a FSP and assume that the FSP to which these states belong have a total of n states and m transitions. Then both the relations $\lesssim$ and $\lessapprox$ may be decided in $O(mn^3)$ time.*

Proof. The correctness follows from proposition 12. As for the time complexity, equation (1) requires $O(n^3)$ time to solve. Since there may be at most m distinct actions, the computation of $\Delta^\dagger$ would require $O(mn^3)$ time (here $O(n^3)$ is the matrix multiplication time). We also know that a natural bisimulation may be computed in $O(mn^2 log(n))$ time since the size of $\Delta^\dagger$ is $O(mn^2)$. And finally the comparison of all tuples for deciding both efficiency prebisimulation and elaboration will not take more than $O(mn^2)$ time. Therefore the total time complexity for the algorithm is $O(mn^3 + n^2 m log(n) + mn^2) = O(mn^3)$. □

4 Conclusion

What we have described is essentially a "global" preorder checking method [6] which may be smoothly integrated into a tool which implements natural bisimulation.

Our algorithm for efficiency prebisimulation reduces the given problem in $O(n^3m)$ time to another problem for which the solution is known and some extra processing whose time complexity is absorbed in the total time complexity of the reduction step. We compare the time complexity of the described method for efficiency prebisimulation, $O(n^3m+n^2mlog(n)+n^2m)$, with that of the weak bisimulation algorithm [13], which is $O(n^\alpha m+n^2mlog(n)), 2<\alpha\leq 3$. For weak bisimulation, the transitive closure of a directed graph having n nodes is computed in $O(n^\alpha)$ time using boolean matrix multiplication for which very elegant algorithms [7] are available. However, in the case of efficiency prebisimulation we are interested not just in finding transitive closure but in the number of τ moves padding each visible action, as well. Therefore it requires a time of $O(n^3)$ to compute both transitive closure as well as the total number of invisible moves.

The entire computation of efficiency prebisimulation and elaboration is done in two steps, where the first step is to reduce the FSP P to another FSP $P^\dagger$ and the second step is to compute the natural bisimulation relation in $P^\dagger$ while comparing pairs of elements.

Rather than verifying a complete system specification against a complete implementation, congruence and precongruence properties may be usefully employed to split up a problem into several smaller individual sub-problems. Verification may then be carried out on the sub-problems.

In the case of CCS, the following result proved in [2] may be used to compute the precongruence relations for CCS processes.

Proposition 14. *For $\sqsubseteq\in\{\lesssim,\lessapprox,\approx\}$, $p\sqsubseteq^c q$, iff for some visible action α not occurring in p or q, $p+\alpha\sqsubseteq q+\alpha$.*

$p\sqsubseteq^c q$ may be determined by using a special action that is not available to the user in the system specification language but is internal to the model checker.

There may exist other methods for tackling state explosion that may be worth exploring. One such is the use of the extension in FSPs as a naming device that abstracts away from complex internal structure and names structurally or behaviourally equal components by the same name. Thus far extensions were used only to distinguish deadlock/termination from other states. But the use of names in extensions may facilitate factoring out parts of systems and thus produce a collection of smaller graphs on which global algorithms may be run locally to check equivalences, congruences, preorders and precongruences.

References

1. Hopcroft, J.E., Aho, A.V., Ullman, J.D.: Design and Analysis of Computer Algorithms. Addison-Wesely, Reading (1974)
2. Arun-Kumar, S., Hennessy, M.: An efficiency preorder for processes. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 152–175. Springer, Heidelberg (1991)
3. Arun-Kumar, S., Hennessy, M.: An efficiency preorder for processes. Acta Informatica 29, 737–760 (1992)

4. Arun-Kumar, S., Natarajan, V.: Conformance: A precongruence close to bisimilarity. In: Structures in Concurrency Theory. Springer Workshops in Computer Science Series, Springer (1995)
5. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. Formal Aspects of Computing 5(1), 1–20 (1993)
6. Cleaveland, R., Sokolsky, O.: Equivalence and preorder checking for finite-state system. In: Handbook of Process Algebra (2001)
7. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progression. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (1987)
8. Gaur, M., Hennessy, M.: Counting the cost in the picalculus (extended abstract). Electronic Notes in Theoretical Computer Science (ENTCS) 229(3), 117–129 (2009)
9. Groote, J.F., Vaandrager, F.W.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 626–638. Springer, Heidelberg (1990)
10. Hennessy, M.: Algebraic Theory of Processes. The MIT Press, Cambridge (1990)
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
12. Hopcroft, J.E., Ullman, J.D.: Intoduction to Automata Theory, Langauages and Computation. Narosa Publishing House (1987)
13. Kanellakis, P.C., Smolka, S.: Ccs expressions, finite state processes and three problem of equivalence. Information and Computation 86(1), 43–68 (1990)
14. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
15. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
16. Milner, R.: Communicating and mobile systems: The π-Calculus. Cambridge University Press (1999)
17. Paige, R., Tarjan, R.E.: Three partition refinement refinement algorithms. SIAM J. Compt. 16(6), 973–989 (1987)
18. Parrow, J., Cleaveland, R., Steffen, B.: The concurrency workbench: A semantic based tool for the verification of the concurrent systems. ACM Transactions of Programming Languages and Systems 15(1) (1993)
19. Sangiorgi, D.: Beyond bisimulation: The "up-to" techniques. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 161–171. Springer, Heidelberg (2006)
20. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (2012)
21. Sangiorgi, D., Milner, R.: The problem of "weak bisimulation up-to". In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 32–46. Springer, Heidelberg (1992)
22. Leiserson, C.E., Cormen, T.H., Rivest, R.L.: Introduction to Algorithms. Eastern Economy Edition. Prentice-Hall India (1990)

Compiling Cooperative Task Management to Continuations

Keiko Nakata and Andri Saar

Institute of Cybernetics at Tallinn University of Technology,
Akadeemia tee 21, 12618 Tallinn, Estonia

Abstract. Although preemptive concurrency models are dominant for multi-threaded concurrency, they may be criticized for the complexity of reasoning because of the implicit context switches. The actor model and cooperative concurrency models have regained attention as they encapsulate the thread of control. In this paper, we formalize a continuation-based compilation of cooperative multitasking for a simple language and prove its correctness.

1 Introduction

In a preemptive concurrency model, threads may be suspended and activated at any time. While preemptive models are dominant for multi-threaded concurrency, they may be criticized for the complexity of reasoning because of the implicit context switches. The programmer often has to resort to low-level synchronization primitives, such as locks, to prevent unwanted context switches. Programs written in such a way tend to be error-prone and are not scalable. The actor model [2] addresses this issue. Actors encapsulate the thread of control and communicate with each other by sending messages. They are also able to call blocking operations such as sleep, await and receive, reminiscent of cooperative multi-tasking. Erlang and Scala actors support actor-based concurrency models.

Creol [8] and ABS [7] combine a message-passing concurrency model and a cooperative concurrency model. In Creol, each object encapsulates a thread of control and objects communicate with each other using asynchronous method calls. Asynchronous method calls, instead of messages-passing, provide a type-safe communication mechanism and are a good match for object-oriented languages [4,3]. ABS generalizes the concurrency model of Creol by introducing *concurrent object groups* [10] as the unit of concurrency. The concurrency model of ABS can be split in two: in one layer, we have local, synchronous and shared-memory communication[1] in one concurrent object group (COG) and on the second layer we have asynchronous message-based concurrency between different concurrent object groups as in Creol. The behavior of one COG is based on the cooperative multitasking of external method invocations and internal

[1] In ABS, different tasks originating from the same object may communicate with each other via fields of the object.

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 95–110, 2013.
DOI: 10.1007/978-3-642-40213-5_7,
© IFIP International Federation for Information Processing 2013

method activations, with concrete scheduling points where a different task may get scheduled. Between different COGs only asynchronous method calls may be used; different COGs have no shared object heap. The order of execution of asynchronous method calls is not specified. The result of an asynchronous call is a future; callers may decide at run-time when to synchronize with the reply from a call. Asynchronous calls may be seen as triggers that spawn new method activations (or tasks) within objects. Every object has a set of tasks that are to be executed (originating from method calls). Among these, at most one task of all the objects belonging to one COG is active; others are suspended and awaiting execution. The concurrency models of Creol and ABS are designed to be suitable in the distributed setting, where one COG executes on its own (virtual) processor in a single node and different COGs may be executed on different nodes in the network.

In this paper, we are interested in the compilation of cooperative multi-tasking into continuations, motivated to execute a cooperative multi-tasking model on the JVM platform, which employs a preemptive model. The basic idea of using continuations to manage the control behavior of the computation has been known from 80's [12,6], and is still considered as a viable technique [9,11,1]. This is particularly so, if the programming language supports first-class continuations, as in the case of Scala, and hence one can obviate manual stack management. The contribution of the paper is a correctness proof of such a compilation scheme. Namely, we create a simplified source language, by extending the While language with (synchronous) procedure calls and operations for cooperative multi-tasking (i.e., blocking operations and creation of new tasks) and define a compilation function from the source language into the target language, which extends While with continuation operations—the target language is sequential. We then prove that the compilation preserves the operational behavior from the source language to the target language.

The remainder of the paper is organized as follows. We define the source language and its operational semantics in the next section, and the target language and its operational semantics in Section 3. In Section 4, we present the compilation function from the source language to the target language and, in Section 5 we prove its correctness. We conclude in Section 6.

2 Source Language

In Figure 1, we define the syntax for the source language. We use the overline notation to denote sequences, with ϵ denoting an empty sequence. It is the While language extended with (local) variable definitions, $\texttt{var}\ x = e$, procedure

$$\begin{array}{rl} S ::= & x := e \mid u := e \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ e\ \texttt{do}\ S \\ \mid & \texttt{var}\ x = e \mid f(\overline{e}) \mid \texttt{await}\ e \mid \texttt{spawn}\ f\ \overline{e} \mid \texttt{return} \end{array}$$

Fig. 1. Syntax of the source language

calls $f(\overline{e})$, the await statement `await` e, creation of a new task `spawn` $f\ \overline{e}$, and the return statement `return`. For simplicity, we syntactically distinguish local variable assignment $x := e$ and global variable assignment $u := e$. The statement `var` $x = e$ defines a (task) local variable x and initializes it with the value of e. The statement $f(\overline{e})$ invokes the procedure f with arguments $\overline{e}$, to be executed within the same task. The procedure does not return the result to the caller, but may store the result in a global variable. The statement `await` e suspends the execution of the current task, which can be resumed when the guard expression e evaluates to true. The statement `spawn` $f\ \overline{e}$ spawns a new task, which executes the body of f with arguments $\overline{e}$. (Hence, in contrast to procedure calls, which are synchronous, `spawn` $f\ \overline{e}$ is like an asynchronous procedure call executed in a new task.) The `return` statement is a runtime construct, not appearing in the source program, and will be explained later.

We assume disjoint supplies of local variables (ranged over by x), global variables (ranged over by u), and procedure names (ranged over by f). We assume a set of (pure) expressions, whose elements are ranged over by e. We assume the set of values to be the integers, non-zero integers counting as truth and zero as falsity. The metavariable v ranges over values. We have two kinds of states – local states and global states. A local state, ranged over by ρ, maps local variables to values; a global state, ranged over by σ, maps global variables to values. We denotes by $\emptyset$ an empty mapping, whose domain is empty. Communication between different tasks is achieved via global variables. For simplicity, we assume a fixed set of global variables. The notation $\rho[x \mapsto v]$ denotes the update of ρ with v at x, when x is in the domain of ρ. If x is not in the domain, it denotes a mapping extension. The notation $\sigma[u \mapsto v]$ denotes similar. We assume given an evaluation function $[\![e]\!]_{(\rho,\sigma)}$, which evaluates e in the local state ρ and the global state σ. We write $(\rho, \sigma) \models e$ and $(\rho, \sigma) \not\models e$ to denote that e is true, resp. false with respect to ρ and σ. A *stack* Π is a non-empty list of local states, whose elements are separated by semicolons. A stack grows leftward, i.e., the leftmost element is the topmost element.

A program P consists of a procedure environment env_F which maps procedure names to pairs of a formal argument list and a statement, and a global state which maps global variables to their initial values. The entry point of the program will be the procedure named `main`.

We define the operational semantics of the source language as a transition system on *configuration*s, in the style of structural operational semantics. A configuration *cfg* consists of an active task identifier n, a global variable mapping σ and a set of tasks Θ. A task has an identifier and may be in one of the three forms: a triple $\langle e, S, \Pi\rangle$, representing a task that is awaiting to be scheduled, where e is the guard expression, S the statement and Π its stack; or, a pair $\langle S, \Pi\rangle$, representing the currently active task; or, a singleton $\langle \Pi\rangle$, representing a terminated task.

$$\begin{array}{lll} \textit{Configuration} & \textit{cfg} & ::= n, \sigma \triangleright \Theta \\ \textit{Task sets} & \Theta & ::= n\langle e, S, \Pi\rangle \mid n\langle S, \Pi\rangle \mid n\langle \Pi\rangle \mid \Theta \parallel \Theta \end{array}$$

The order of tasks in the task set is irrelevant: the parallel operator $\|$ is commutative and associative. Formally, we assume the following structural equivalence:

$$\Theta \equiv \Theta \qquad \Theta \parallel \Theta' \equiv \Theta' \parallel \Theta \qquad \Theta \parallel (\Theta' \parallel \Theta'') \equiv (\Theta \parallel \Theta') \parallel \Theta''$$

Transition rules in the semantics are in the form $\mathrm{env}_F \vdash cfg \to cfg'$, shown in Figure 2. The first two rules (S-Cong and S-Equiv) deal with congruence and structural equivalence. The rules for assignment, `skip`, if-then-else and while are self-explanatory. For instance, in the rule S-Assign-Local, the task is of the form $n\langle S, \Pi' \rangle$ where $S = x := e$ and $\Pi' = \rho; \Pi$. Note that the topmost element of the stack Π is the current local state. The rules for sequential composition may deserve some explanation. If the first statement S_1 suspends guarded by e in the stack Π' with the residual statement S_1' to be run when resumed, then the entire statement $S_1; S_2$ suspends in $\langle e, S_1'; S_2, \Pi' \rangle$, where the residual statement now contains the second statement S_2 (S-Seq-Grd). If S_1 terminates in Π', then S_2 will run next in Π' (S-Seq-Fin). Otherwise, S_1 transfers to S_1' with the stack Π', so that $S_1; S_2$ transfers to $S_1'; S_2$ with the same stack (S-Seq-Step). The await statement immediately suspends (S-Await) the currently active task, enabling us to switch to some other task in accordance to the scheduling rules. An example of the await statement (and the scheduling rules) at work can be found in the example in Figure 3. The statement $\texttt{spawn}\ f\ \overline{e}$ creates a new task $n'\langle \texttt{true}, S, [\overline{x} \mapsto \overline{v}] \rangle$ with n' a fresh identifier (S-Spawn). The caller task continues to be active. The newly created task is suspended, guarded by `true`, and may get scheduled at scheduling points by the scheduling rules (see below). Procedure invocation $f(\overline{e})$ evaluates the arguments $\overline{e}$ in the current state, pushes into the stack the local state $[\overline{x} \mapsto \overline{v}]$, mapping the formal parameters to the actual arguments, and transfers to $S; \texttt{return}$, where S is the body of f (S-Call). The `return` statement pops the topmost element from the stack (S-Return). The local variable definition $\texttt{var}\ x = e$ extends the current local state with the newly defined variable and initializes it with the value of e (S-Var).

The last three rules deal with scheduling. If the current active task has terminated, then a new task whose guard evaluates to true is chosen to be active (S-Sched-Fin). When the active task suspends, a scheduling point is reached. The rule (S-Sched-Same) considers the case in which the same task is scheduled; the rule (S-Sched-Other) considers the case in which a different task is scheduled.

As an example, we will look at a program containing one global variable u with the initial value 0 and the following procedures:

$$\begin{aligned} \mathsf{f} &\mapsto u := 1 \\ \mathsf{main} &\mapsto u := 3; \texttt{spawn}\ \mathsf{f}\ \epsilon; \texttt{await}\ u = 1; u := 2 \end{aligned}$$

A detailed step, showing the full derivation, can be seen in Figure 4. A full execution trace, showing all intermediate configurations, is shown in Figure 3.

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright \Theta' \rightarrow n', \sigma' \triangleright \Theta''}{\mathrm{env}_F \vdash n, \sigma \triangleright \Theta \parallel \Theta' \rightarrow n', \sigma' \triangleright \Theta \parallel \Theta''} \ \textsc{S-Cong}$$

$$\frac{\Theta \equiv \Theta' \quad \mathrm{env}_F \vdash n, \sigma \triangleright \Theta' \rightarrow n', \sigma' \triangleright \Theta''' \quad \Theta'' \equiv \Theta'''}{\mathrm{env}_F \vdash n, \sigma \triangleright \Theta \rightarrow n', \sigma' \triangleright \Theta''} \ \textsc{S-Equiv}$$

$$\frac{x \in \mathsf{dom}\,\rho}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle x := e, \rho; \Pi\rangle \rightarrow n, \sigma \triangleright n\langle \rho[x \mapsto [\![e]\!]_{(\rho,\sigma)}]; \Pi\rangle} \ \textsc{S-Assign-Local}$$

$$\frac{u \in \mathsf{dom}\,\sigma}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle u := e, \Pi\rangle \rightarrow n, \sigma[u \mapsto [\![e]\!]_{(\rho,\sigma)}] \triangleright n\langle \Pi\rangle} \ \textsc{S-Assign-Global}$$

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1, \Pi\rangle \rightarrow n', \sigma' \triangleright n\langle e, S_1', \Pi'\rangle \parallel \Theta}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1; S_2, \Pi\rangle \rightarrow n', \sigma' \triangleright n\langle e, S_1'; S_2, \Pi'\rangle \parallel \Theta} \ \textsc{S-Seq-Grd}$$

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1, \Pi\rangle \rightarrow n', \sigma' \triangleright n\langle \Pi'\rangle \parallel \Theta}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1; S_2, \Pi\rangle \rightarrow n', \sigma' \triangleright n\langle S_2, \Pi'\rangle \parallel \Theta} \ \textsc{S-Seq-Fin}$$

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1, \Pi\rangle \rightarrow n', \sigma' \triangleright n\langle S_1', \Pi'\rangle \parallel \Theta}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1; S_2, \Pi\rangle \rightarrow n', \sigma' \triangleright n\langle S_1'; S_2, \Pi'\rangle \parallel \Theta} \ \textsc{S-Seq-Step}$$

$$\frac{(\rho, \sigma) \models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \rho; \Pi\rangle \rightarrow n, \sigma \triangleright n\langle S_1, \rho; \Pi\rangle} \ \textsc{S-If-True}$$

$$\frac{(\rho, \sigma) \not\models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \rho; \Pi\rangle \rightarrow n, \sigma \triangleright n\langle S_2, \rho; \Pi\rangle} \ \textsc{S-If-False}$$

$$\frac{(\rho, \sigma) \models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{while}\ e\ \mathtt{do}\ S, \rho; \Pi\rangle \rightarrow n, \sigma \triangleright n\langle S; \mathtt{while}\ e\ \mathtt{do}\ S, \rho; \Pi\rangle} \ \textsc{S-While-True}$$

$$\frac{(\rho, \sigma) \not\models e}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{while}\ e\ \mathtt{do}\ S, \rho; \Pi\rangle \\ \quad \rightarrow n, \sigma \triangleright n\langle \rho; \Pi\rangle\end{array}} \ \textsc{S-While-False} \qquad \frac{}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{skip}, \Pi\rangle \\ \quad \rightarrow n, \sigma \triangleright n\langle \Pi\rangle\end{array}} \ \textsc{S-Skip}$$

$$\frac{\mathrm{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{(\rho,\sigma)} \quad n' \text{ is fresh}}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{spawn}\ f\ \overline{e}, \rho; \Pi\rangle \\ \quad \rightarrow n, \sigma \triangleright n\langle \rho; \Pi\rangle \parallel n'\langle \mathtt{true}, S, [\overline{x} \mapsto \overline{v}]\rangle\end{array}} \ \textsc{S-Spawn} \qquad \frac{}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{await}\ e, \Pi\rangle \\ \quad \rightarrow n, \sigma \triangleright n\langle e, \mathtt{skip}, \Pi\rangle\end{array}} \ \textsc{S-Await}$$

$$\frac{x \notin \mathsf{dom}\,\rho}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{var}\ x = e, \rho; \Pi\rangle \\ \quad \rightarrow n, \sigma \triangleright n\langle \rho[x \mapsto [\![e]\!]_{(\rho,\sigma)}]; \Pi\rangle\end{array}} \ \textsc{S-Var} \qquad \frac{\mathrm{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{(\rho,\sigma)}}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle f(\overline{e}), \rho; \Pi\rangle \\ \quad \rightarrow n, \sigma \triangleright n\langle S; \mathtt{return}, [\overline{x} \mapsto \overline{v}]; \rho; \Pi\rangle\end{array}} \ \textsc{S-Call}$$

$$\frac{}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{return}, \rho; \Pi\rangle \\ \quad \rightarrow n, \sigma \triangleright n\langle \Pi\rangle\end{array}} \ \textsc{S-Return} \qquad \frac{(\rho, \sigma) \models e}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \Pi\rangle \parallel n'\langle e, S, \rho; \Pi'\rangle \\ \quad \rightarrow n', \sigma \triangleright n'\langle S, \rho; \Pi'\rangle \parallel n\langle \Pi\rangle\end{array}} \ \textsc{S-Sched-Fin}$$

$$\frac{(\rho, \sigma) \models e}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle e, S, \rho; \Pi\rangle \\ \quad \rightarrow n, \sigma \triangleright n\langle S, \rho; \Pi\rangle\end{array}} \ \textsc{S-Sched-Same} \qquad \frac{(\rho', \sigma) \models e'}{\begin{array}{l}\mathrm{env}_F \vdash n, \sigma \triangleright n\langle e, S, \rho\rangle \parallel n'\langle e', S', \rho'\rangle \\ \quad \rightarrow n', \sigma \triangleright n'\langle S', \rho'\rangle \parallel n\langle e, S, \rho\rangle\end{array}} \ \textsc{S-Sched-Other}$$

Fig. 2. Semantics of the source language

$$
\begin{aligned}
\mathrm{env}_F \vdash\ & \mathsf{main}, [u \mapsto 0] \triangleright \mathsf{main}\langle u := 3; \texttt{spawn f } \epsilon; \texttt{await } u = 1; u := 2, \emptyset\rangle \\
\to\ & \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{spawn f } \epsilon; \texttt{await } u = 1; u := 2, \emptyset\rangle \\
\to\ & \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{await } u = 1; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle \\
\to\ & \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle \\
\to\ & \mathsf{f}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle u := 1, \emptyset\rangle \\
\to\ & \mathsf{f}, [u \mapsto 1] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \emptyset\rangle \\
\to\ & \mathsf{main}, [u \mapsto 1] \triangleright \mathsf{main}\langle \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \emptyset\rangle \\
\to\ & \mathsf{main}, [u \mapsto 1] \triangleright \mathsf{main}\langle u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \emptyset\rangle \\
\to\ & \mathsf{main}, [u \mapsto 2] \triangleright \mathsf{main}\langle \emptyset\rangle \parallel \mathsf{f}\langle \emptyset\rangle
\end{aligned}
$$

Fig. 3. The full execution trace of the example program

$$
\dfrac{\dfrac{\dfrac{}{\begin{array}{c}\mathrm{env}_F \vdash \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{await } u = 1, \emptyset\rangle \\ \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}, \emptyset\rangle\end{array}}\ \textsc{S-Await}}{\begin{array}{c}\mathrm{env}_F \vdash \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{await } u = 1; u := 2, \emptyset\rangle \\ \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle\end{array}}\ \textsc{S-Seq-Grd}}{\begin{array}{c}\mathrm{env}_F \vdash \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{await } u = 1; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle \\ \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle\end{array}}\ \textsc{S-Cong}
$$

Fig. 4. Example of a derivation in the source language

3 Target Language

We proceed to the target language. In Figure 5, we present the syntax of the target language. Expressions of the target language contain, besides pure expressions, *continuations* $[S, \Pi]$, which are pairs of a statement S and a stack Π, and support for *guarded (multi)sets*: collections which contain pairs of an expression and a value. The expression stored with each element is called a *guard expression*, and is evaluated when we query the set: only elements whose guard expressions hold may be returned. There are five expressions in the language to work with guarded sets: an empty set $\emptyset$, checking whether a set is empty ($\texttt{isEmpty}\ e_s$), adding an element ($\texttt{add}\ e_s\ e_g\ e$), fetching an element ($\texttt{get}\ e_s$) and removing an element ($\texttt{del}\ e_s\ e$).

Similar to the source language, for the target language we extend While with local variable definitions and procedure calls. We also add delimited control operators, $\texttt{shift}\ k\ \{S\}$, $\texttt{reset}\ \{S\}$, $\texttt{invoke}\ k$ [5]. The statement $\texttt{shift}\ k\ \{S\}$ captures the rest of the computation, or continuation, up to the closest surrounding $\texttt{reset}\ \{\}$, binds it to k, and proceeds to execute S. In $\texttt{shift}\ k\ \{S\}$, k is a binding occurrence, whose scope is S. Hence, the statement $\texttt{reset}\ \{S\}$ delimits the captured continuation. The statement $\texttt{invoke}\ k$ invokes, or jumps into, the continuation bound to the variable k. The statement $\texttt{R}\ \{S\}$, where $\texttt{R}$ is a new constant, is a runtime construct to be explained later.

The target language is sequential and, unlike the source language, it contains no explicit support for parallelism. Instead, we provide building blocks – continuations and guarded sets – that are used to switch between tasks and implement an explicit scheduler in Section 4.

$$
\begin{array}{lll}
\textit{Expressions} & e ::= & \ldots \\
& \mid & [S, \Pi] \mid \emptyset \mid \texttt{isEmpty}\ e \mid \texttt{add}\ e_1\ e_2\ e_3 \mid \texttt{del}\ e_1\ e_2 \mid \texttt{get}\ e \\
\textit{Statements} & S ::= & x := e \mid u := e \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ e\ \texttt{do}\ S \\
& \mid & \texttt{var}\ x = e \mid f(\overline{e}) \mid \texttt{reset}\ \{S\} \mid \texttt{shift}\ k\ \{S\} \mid \texttt{invoke}\ k \\
& \mid & \texttt{return} \mid \texttt{R}\ \{S\}
\end{array}
$$

Fig. 5. Syntax of the target language

We assume given an evaluation function $[\![e]\!]_{(\Pi,\sigma)}$ for pure expressions, which evaluates e with respect to the stack Π (the evaluation only looks at the current local state, which is the topmost element of Π) and the global state σ. In Figure 6, the evaluation function is extended to operations for guarded sets.

We define an operational semantics for the target language as a reduction semantics over configurations, using evaluation contexts. A configuration $\langle S, \Pi, \sigma \rangle$ is a triple of a statement S, a stack Π and a global state σ.

Evaluation contexts E are statements with a hole, specifying where the next reduction may occur. They are defined by

$$E ::= [] \mid E; S \mid \texttt{R}\ \{E\}$$

We denote by $E[S]$ the statement obtained by placing S in the hole of E.

We define basic reduction rules in Figure 6. `reset` $\{S\}$ inserts a marker † into the stack, just below the current state, and reduces to `R` $\{S\}$ to continue the execution of S (T-Reset). We use a marker to delimit the portion of the stack captured by `shift` and to align the stack when exiting from `R` $\{\}$. The runtime construct `R` $\{\}$ is used to record that the marker has been set. `shift` k $\{S\}$ captures the rest of the execution up to and including the closest surrounding `R` $\{\}$ together with the corresponding portion of the stack, binds it to a fresh variable k' in the local state, and continues with the statement S' obtained by substituting k by k' in S (T-Shift). The surrounding `R` $\{\}$ is kept intact. F is an evaluation context that does not intersect `R` $\{\}$, formally,

$$F ::= [] \mid F; S$$

Note that `shift` k $\{S\}$ captures the stack up to and including the topmost †, which has been inserted by the closest surrounding reset. Once the body of `R` $\{\}$ terminates, i.e., reduces to `skip`, then we remove the `R` $\{\}$ and pop the stack until the topmost †, but leaving the state just above † in the stack (T-R). `invoke` k invokes the continuation bound to k (T-Invoke). Namely, if k is bound to $[S, \Pi']$ in the local or global state, then the statement reduces to $S;$`return` and the stack Π' is pushed into the current stack. S must be necessarily of the form `R` $\{S'\}$, where S' does not contain `R`, and Π' contains exactly one † at the bottom. When exiting from the `R` $\{\}$, the state immediately above † in Π' will be left in the stack, which is popped by the trailing `return`. An example of how to capture and invoke a continuation is shown in Figure 7. In the example, we assume that the variables u and u' are global.

$$\begin{aligned}
[\![\texttt{add}\ e_s\ e_g\ e]\!]_{(\Pi,\sigma)} &= [\![e_s]\!]_{(\Pi,\sigma)} \cup (e_g, [\![e]\!]_{(\Pi,\sigma)}) \\
[\![\texttt{get}\ e_s]\!]_{(\Pi,\sigma)} &= v \text{ when } \exists e_g \exists v.\ (e_g, v) \in [\![e_s]\!]_{(\Pi,\sigma)} \wedge [\![e_g]\!]_{(\Pi,\sigma)} = \texttt{true} \\
[\![\texttt{del}\ e_s\ e]\!]_{(\Pi,\sigma)} &= [\![e_s]\!]_{(\Pi,\sigma)} \backslash (e_g, [\![e]\!]_{(\Pi,\sigma)}) \text{ when } \exists e_g.\ (e_g, [\![e]\!]_{(\Pi,\sigma)}) \in [\![e_s]\!]_{(\Pi,\sigma)} \\
[\![\texttt{isEmpty}\ e_s]\!]_{(\Pi,\sigma)} &= \begin{cases} \texttt{true} & \text{if } [\![e_s]\!]_{(\Pi,\sigma)} = \emptyset \\ \texttt{false} & \text{otherwise} \end{cases}
\end{aligned}$$

$$\frac{}{\mathrm{env}_F \vdash \langle \texttt{reset}\ \{S\}, \rho; \Pi, \sigma\rangle \to \langle \texttt{R}\ \{S\}, \rho\dagger; \Pi, \sigma\rangle}\ \text{T-Reset}$$

$$\frac{\Pi \text{ does not contain } \dagger \quad k' \text{ fresh} \quad S' \text{ is obtained from S by replacing } k \text{ with } k'}{\begin{array}{l}\mathrm{env}_F \vdash \langle \texttt{R}\ \{F[\texttt{shift}\ k\ \{S\}]\}, \rho; \Pi\dagger; \Pi', \sigma\rangle \\ \quad \to \langle \texttt{R}\ \{S'\}, \rho[k' \mapsto [\texttt{R}\ \{F[\texttt{skip}]\}, \rho; \Pi\dagger]]; \Pi\dagger; \Pi', \sigma\rangle\end{array}}\ \text{T-Shift}$$

$$\frac{\Pi \text{ does not contain } \dagger}{\mathrm{env}_F \vdash \langle \texttt{R}\ \{\texttt{skip}\}, \Pi; \rho\dagger; \Pi', \sigma\rangle \mapsto \langle \texttt{skip}, \rho; \Pi', \sigma\rangle}\ \text{T-R}$$

$$\frac{[\![k]\!]_{(\Pi,\sigma)} = [S, \Pi']}{\mathrm{env}_F \vdash \langle \texttt{invoke}\ k, \Pi, \sigma\rangle \to \langle S; \texttt{return}, \Pi'; \Pi, \sigma\rangle}\ \text{T-Invoke}$$

$$\frac{}{\mathrm{env}_F \vdash \langle \texttt{skip}; S, \Pi, \sigma\rangle \to \langle S, \Pi, \sigma\rangle}\ \text{T-Skip}$$

$$\frac{\mathrm{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{(\Pi,\sigma)}}{\mathrm{env}_F \vdash \langle f(\overline{e}), \Pi, \sigma\rangle \to \langle S; \texttt{return}, [\overline{x} \mapsto \overline{v}]; \Pi, \sigma\rangle}\ \text{T-Call}$$

$$\frac{}{\mathrm{env}_F \vdash \langle \texttt{return}, \rho; \Pi, \sigma\rangle \to \langle \texttt{skip}, \Pi, \sigma\rangle}\ \text{T-Return}$$

$$\frac{(\Pi, \sigma) \models e}{\mathrm{env}_F \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \Pi, \sigma\rangle \to \langle S_1, \Pi, \sigma\rangle}\ \text{T-If-True}$$

$$\frac{(\Pi, \sigma) \not\models e}{\mathrm{env}_F \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \Pi, \sigma\rangle \to \langle S_2, \Pi, \sigma\rangle}\ \text{T-If-False}$$

$$\frac{(\Pi, \sigma) \models e}{\mathrm{env}_F \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \Pi, \sigma\rangle \to \langle S; \texttt{while}\ e\ \texttt{do}\ S, \Pi, \sigma\rangle}\ \text{T-While-True}$$

$$\frac{(\Pi, \sigma) \not\models e}{\mathrm{env}_F \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \Pi, \sigma\rangle \to \langle \texttt{skip}, \Pi, \sigma\rangle}\ \text{T-While-False}$$

$$\frac{x \notin \mathsf{dom}\,\rho}{\mathrm{env}_F \vdash \langle \texttt{var}\ x = e, \rho; \Pi, \sigma\rangle \to \langle \texttt{skip}, \rho[x \mapsto [\![e]\!]_{(\rho;\Pi,\sigma)}]; \Pi, \sigma\rangle}\ \text{T-Var}$$

$$\frac{x \in \mathsf{dom}\,\rho}{\mathrm{env}_F \vdash \langle x := e, \rho; \Pi, \sigma\rangle \to \langle \texttt{skip}, \rho[x \mapsto [\![e]\!]_{(\rho;\Pi,\sigma)}]; \Pi, \sigma\rangle}\ \text{T-Assign-Local}$$

$$\frac{u \in \mathsf{dom}\,\sigma}{\mathrm{env}_F \vdash \langle u := e, \Pi, \sigma\rangle \to \langle \texttt{skip}, \Pi, \sigma[u \mapsto [\![e]\!]_{(\Pi,\sigma)}]\rangle}\ \text{T-Assign-Global}$$

Fig. 6. Semantics of the target language

$$
\begin{aligned}
&\text{env}_F \vdash \langle \texttt{reset}\ \{u' := 1; \texttt{shift}\ k\ \{u := k\}; u' := 0\}; \texttt{invoke}\ u, \emptyset, [u \mapsto 0; u' \mapsto 0]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{u' := 1; \texttt{shift}\ k\ \{u := k\}; u' := 0\}; \texttt{invoke}\ u, \emptyset\dagger, [u \mapsto 0; u' \mapsto 0]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{\texttt{shift}\ k\ \{u := k\}; u' := 0\}; \texttt{invoke}\ u, \emptyset\dagger, [u \mapsto 0; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{u := k'\}; \texttt{invoke}\ u, [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]]\dagger, [u \mapsto 0; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{\texttt{skip}\}; \texttt{invoke}\ u, [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]]\dagger, [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{invoke}\ u, [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{\texttt{skip}; u' := 0\}; \texttt{return}, \emptyset\dagger; \\
&\qquad\quad [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{u' := 0\}; \texttt{return}, \emptyset\dagger; [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{\texttt{skip}\}; \texttt{return}, \emptyset\dagger; [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 0]\rangle \\
&\quad \mapsto \langle \texttt{return}, \emptyset; [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 0]\rangle \\
&\quad \mapsto \langle \texttt{skip}, [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 0]\rangle
\end{aligned}
$$

Fig. 7. Capturing and invoking a continuation

Procedure call $f(\overline{e})$ reduces to S; `return` where S is the body of the procedure f, and pushes a local state $[\overline{x} \mapsto \overline{v}]$, binding procedure's formal arguments to actual arguments, into the stack (T-Call). The trailing `return` ensures that, once the execution of S; `return` terminates, the stack is aligned to the original Π. `return` pops the topmost element from the stack (T-Return). The remaining rules are self-explanatory.

Given the basic reduction rules, we now define a standard reduction, denoted by $\mapsto$, by

$$\frac{\text{env}_F \vdash \langle S, \Pi, \sigma\rangle \rightarrow \langle S', \Pi', \sigma'\rangle}{\text{env}_F \vdash \langle E[S], \Pi, \sigma\rangle \mapsto \langle E[S'], \Pi', \sigma'\rangle}$$

stating that the configuration $\langle S, \Pi, \sigma\rangle$ standard reduces to $\langle S', \Pi', \sigma'\rangle$ if there exist an evaluation context E and statement S_0 and S_0' such that $S = E[S_0]$ and $S' = E[S_0']$ and $\text{env}_F \vdash \langle S_0, \Pi, \sigma\rangle \rightarrow \langle S_0', \Pi', \sigma'\rangle$. The standard reduction is deterministic.

4 Compilation

When compiling a program P into the target language, we compile expressions and statements according to the scheme shown in Figure 8. Expressions are translated into the target language as-is, and statements that have a corresponding equivalent in the target language are also translated in a straightforward manner. The two statements that have no direct correspondence in the target language are `await` and `spawn`. We look at how these statements are translated and how they interact with the scheduler later.

The central idea of the compilation scheme is to use continuations to handle the suspension of tasks, and have an explicit *scheduler* (for brevity, in the examples we use *Sched* to denote the scheduler), shown in Figure 9. The control will pass to the scheduler every time a task either suspends or finishes, and the

$$
\begin{aligned}
[\![e]\!] &= e \\
[\![x := e]\!] &= x := [\![e]\!] \\
[\![u := e]\!] &= u := [\![e]\!] \\
[\![\texttt{skip}]\!] &= \texttt{skip} \\
[\![S_1; S_2]\!] &= [\![S_1]\!]; [\![S_2]\!] \\
[\![\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2]\!] &= \texttt{if } [\![e]\!] \texttt{ then } [\![S_1]\!] \texttt{ else } [\![S_2]\!] \\
[\![\texttt{while } e \texttt{ do } S]\!] &= \texttt{while } [\![e]\!] \texttt{ do } [\![S]\!] \\
[\![\texttt{var } x = e]\!] &= \texttt{var } x = [\![e]\!] \\
[\![\texttt{await } e]\!] &= \texttt{shift } k \ \{T := \texttt{add } T \ [\![e]\!] \ k\}; \texttt{skip} \\
[\![f(\overline{e})]\!] &= f_S(\overline{[\![e]\!]}) \\
[\![\texttt{spawn } f \ \overline{e}]\!] &= f_A(\overline{[\![e]\!]})
\end{aligned}
$$

Fig. 8. Compilation of source programs

$$
\mathit{Sched} = \texttt{while } (\neg\texttt{isEmpty } T) \texttt{ do reset } \{k := \texttt{get } T; T := \texttt{del } T \ k; \texttt{invoke } k\}
$$

Fig. 9. Scheduler

scheduler will pick up a new task to execute. During runtime, we also use a global variable T, which we assume not to be used by the program to be compiled. The global variable T stores the *task set*, corresponding to Θ in the source semantics, that contains all the tasks in the system. The tasks are stored as continuations with guard expressions.

The scheduler loops until the task set is empty (all tasks have terminated), in each iteration picking a continuation from T where the guard expression evaluates to `true`, removing it from T and then invoking the continuation using `invoke` k. The body of the scheduler is wrapped in a `reset`, guaranteeing that when a task suspends, the capture will be limited to the end of the current task. After the execution is completed – either by suspension or by just finishing the work – the control comes back to the scheduler.

Suspension (`await` e in the source language) is compiled to a `shift` statement. When evaluating the statement, the original computation until the end of the enclosing `R` $\{\}$ will be captured and stored in the continuation k, and the original program is replaced with the body of the `shift`. The enclosing `R` $\{\}$ guarantees that we capture only the statements up until the end of the current task, thus providing a facility to proceed with the execution of the task later. The body of the `shift` statement simply takes the captured continuation, k, and adds it to the global task set, with the appropriate guard expression. After adding the continuation to the task set, the control passes back to the scheduler.

$$\llbracket n, \sigma \rhd n\langle \Pi \rangle \parallel \Theta \rrbracket = \langle Sched, \emptyset, \sigma[T \mapsto \llbracket \Theta \rrbracket_2] \rangle$$
$$\llbracket n, \sigma \rhd n\langle e, S, \Pi \rangle \parallel \Theta \rrbracket = \langle Sched, \emptyset, \sigma[T \mapsto \llbracket \Theta \rrbracket_2 \cup \llbracket n\langle e, S, \rho \rangle \rrbracket_2] \rangle$$
$$\llbracket n, \sigma \rhd n\langle S, \Pi \rangle \parallel \Theta \rrbracket = \langle \mathtt{R}\ \{\mathtt{R}\ \{\llbracket S \rrbracket\}; \mathtt{return}\}; Sched, \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto \llbracket \Theta \rrbracket_2] \rangle$$
$$\llbracket \Theta \parallel \Theta' \rrbracket_2 = \llbracket \Theta \rrbracket_2 \cup \llbracket \Theta' \rrbracket_2$$
$$\llbracket n\langle e, S, \Pi \rangle \rrbracket_2 = \{(\llbracket e \rrbracket, [\mathtt{R}\ \{\mathtt{skip}; \llbracket S \rrbracket\}, \Pi\dagger])\}$$
$$\llbracket n\langle \Pi \rangle \rrbracket_2 = \emptyset$$

Fig. 10. Compilation of configurations

Procedures in env_F will get translated into two different procedures for synchronous and asynchronous calls, as follows:

$$f_S \mapsto \llbracket S \rrbracket$$
$$f_A \mapsto \mathtt{reset}\ \{\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ \mathtt{true}\ k\}; \llbracket S \rrbracket\}$$

When making an asynchronous call, the body of the procedure will be immediately captured in a continuation, added to the global task set, and the control passes back to the invoker via the usual synchronous call mechanism.

The entry point of a program in the source language, main, is a regular procedure and will get translated according to the usual rules into two procedures, main_A and main_S. In the target language, we must invoke the scheduler, and thus we use a different entry point:

$$T := \emptyset; \mathsf{main}_A(); Sched$$

After initializing the task set to be empty, the first statement will add an asynchronous call to the original entry point of the program, and passes control to the scheduler. As there is only one task in the task set – the task that will invoke the original entry point – the scheduler will immediately proceed with that.

5 Correctness

In this section, we prove that our compilation scheme is correct in the sense that it preserves the operational behavior from the source program into the (compiled) target program. Specifically, we prove that reductions in the source language are simulated by corresponding reductions in the target language. To do so, we extend the compilation scheme to configurations in Figure 10.

The compilation scheme for configurations follows the idea of the compilation scheme detailed in Section 4. We have two compilation functions: $\llbracket \cdot \rrbracket_2$, which generates a task set from Θ, and $\llbracket \cdot \rrbracket$, which generates a configuration in the target semantics.

Every suspended task in the task set Θ is compiled to a pair consisting of the compiled guard expression and a continuation that has been constructed from

$$\frac{\rho\ x = [S, \Pi']}{\text{env}_F \vdash \langle \texttt{invoke}\ x, \rho; \Pi, \sigma\rangle \rightarrow \langle S; \texttt{return}, \Pi'; \rho\backslash x; \Pi, \sigma\rangle}\ \text{T-InvokeOnce}$$

Fig. 11. Alternative rule for `invoke`

the original statement and stack. The statement is wrapped in a `R` {} block and we prepend a `skip` statement, just as it would happen when a continuation is captured in the target language.

If the active task is finished or is suspended (but no new task has been scheduled yet), the generated configuration will immediately contain the scheduler. If the task has suspended, the task is compiled according to the previously described scheme and appended to T. Active tasks are wrapped in two `R` {} blocks and the stack Π is concatenated on top of the local state of the scheduler.

When the scheduler invokes a continuation k, the continuation will stay in the local state of the scheduler until control comes back to the scheduler. This is unnecessary, as the value is never used after it has been invoked; furthermore, the variable is immediately assigned a new value after control passes back to the scheduler. Thus, as an optimization, we may switch to an alternative reduction rule for `invoke` k, which only allows a continuation to be used once, T-InvokeOnce, shown in Figure 11. Although the behavior of the program is equivalent under both versions, using the one-shot version also allows us to state the correctness theorem in a more concise and straightforward manner, as the local state of the scheduler will always be empty when we are currently executing some task. In the proof, we assume this rule to be used instead of the original T-Invoke rule.

The following lemma states that the compilation of statements is compositional with respect to evaluation contexts, where evaluation contexts for the source language are defined inductively by

$$K := [] \mid K; S.$$

Lemma 1.

$$[\![K[S]]\!] = [\![K]\!]\,[[\![S]\!]]$$

Proof. By induction on the structure of K. □

The correctness theorem below states that a one-step reduction in the source language is simulated by multiple-step reductions in the target language.

As an example, in Figure 12 we show the compiled form for both the initial and final configurations shown for the step in Figure 4 and in Figure 13, we show how to reach the compiled equivalent of the configuration in multiple steps in the target semantics.

Theorem 1. *For all configurations cfg_S and cfg'_S such that*

$$\text{env}_F \vdash cfg_S \rightarrow cfg'_S$$

$$\begin{aligned}
&[\![\mathsf{main}, [u \mapsto 0] \triangleright \mathsf{main}\langle \texttt{await } u = 1; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle]\!] \\
&\quad = \langle \texttt{R } \{\texttt{R } \{\texttt{shift } k \ \{T := \texttt{add } T \ (u = 1) \ k\}; \texttt{skip}; u := 2\}; \texttt{return}\}; \mathit{Sched}, \emptyset\dagger; \emptyset\dagger, \\
&\qquad\quad [u \mapsto 0; T \mapsto \{(\texttt{true}, [\texttt{R } \{\texttt{skip}; u := 1\}, \emptyset\dagger])\}]\rangle \\
&[\![\mathsf{main}, [u \mapsto 0] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle]\!] \\
&\quad = \langle \mathit{Sched}, \emptyset, [u \mapsto 0; T \mapsto \{(u = 1, [\texttt{R } \{\texttt{skip}; \texttt{skip}; u := 2\}, \emptyset\dagger]), (\texttt{true}, [\texttt{R } \{\texttt{skip}; u := 1\}, \emptyset\dagger])\}]\rangle
\end{aligned}$$

Fig. 12. Example of compiling a configuration

$$\begin{aligned}
&\langle \texttt{R } \{\texttt{R } \{\texttt{shift } k \ \{T := \texttt{add } T \ (u = 1) \ k\}; \texttt{skip}; u := 2\}; \texttt{return}\}; \mathit{Sched}, \emptyset\dagger; \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\texttt{true}, [\texttt{R } \{\texttt{skip}; u := 1\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \texttt{R } \{\texttt{R } \{T := \texttt{add } T \ (u = 1) \ k'\}; \texttt{return}\}; \mathit{Sched}, [k' \mapsto [\texttt{R } \{\texttt{skip}; \texttt{skip}; u := 2\}, \emptyset\dagger]]\dagger; \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\texttt{true}, [\texttt{R } \{\texttt{skip}; u := 1\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \texttt{R } \{\texttt{R } \{\texttt{skip}\}; \texttt{return}\}; \mathit{Sched}, [k' \mapsto [\texttt{R } \{\texttt{skip}; \texttt{skip}; u := 2\}, \emptyset\dagger]]\dagger; \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\texttt{true}, [\texttt{R } \{\texttt{skip}; u := 1\}, \emptyset\dagger]), (u = 1, [\texttt{R } \{\texttt{skip}; \texttt{skip}; u := 2\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \texttt{R } \{\texttt{return}\}; \mathit{Sched}, [k' \mapsto [\texttt{R } \{\texttt{skip}; \texttt{skip}; u := 2\}, \emptyset\dagger]]; \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\texttt{true}, [\texttt{R } \{\texttt{skip}; u := 1\}, \emptyset\dagger]), (u = 1, [\texttt{R } \{\texttt{skip}; \texttt{skip}; u := 2\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \texttt{R } \{\texttt{skip}\}; \mathit{Sched}, \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\texttt{true}, [\texttt{R } \{\texttt{skip}; u := 1\}, \emptyset\dagger]), (u = 1, [\texttt{R } \{\texttt{skip}; \texttt{skip}; u := 2\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \mathit{Sched}, \emptyset, [u \mapsto 0; T \mapsto \{(\texttt{true}, [\texttt{R } \{\texttt{skip}; u := 1\}, \emptyset\dagger]), (u = 1, [\texttt{R } \{\texttt{skip}; \texttt{skip}; u := 2\}, \emptyset\dagger])\}]\rangle
\end{aligned}$$

Fig. 13. Reduction of the compiled configuration

holds, then the following must also hold:

$$[\![\mathrm{env}_F]\!] \vdash [\![\mathit{cfg}_S]\!] \mapsto^+ [\![\mathit{cfg}'_S]\!].$$

Proof. By induction over the derivation, analyzing the step taken. The possible steps have one of the following forms:

– Case

$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S, \Pi\rangle \parallel \Theta \to n', \sigma' \triangleright n\langle \Pi'\rangle \parallel \Theta'$$

Rules matching this pattern are S-Assign-Local, S-Assign-Global, S-While-False, S-Spawn, S-Return, S-Var. As a representative example, we will look at S-Spawn in detail.

$$\frac{\mathrm{env}_F \ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{(\rho,\sigma)} \quad n' \text{ is fresh}}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{spawn } f \ \overline{e}, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle \rho; \Pi\rangle \parallel n'\langle \texttt{true}, S, [\overline{x} \mapsto \overline{v}]\rangle}$$

In this case, the source and target configurations are compiled to:

$$\begin{aligned}
[\![\mathit{cfg}_S]\!] &= \langle \texttt{R } \{\texttt{R } \{f_A(\overline{e})\}; \texttt{return}\}; \mathit{Sched}, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
[\![\mathit{cfg}'_S]\!] &= \langle \mathit{Sched}, \emptyset, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\texttt{true}, [\texttt{R } \{\texttt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle
\end{aligned}$$

Let the bottommost element of Π be ρ', where $\rho = \rho'$ if Π is empty. The compiled source configuration will reduce as follows:

$$
\begin{aligned}
&[\![\mathrm{env}_F]\!] \vdash \langle \mathtt{R}\ \{\mathtt{R}\ \{f_A(\overline{e})\}; \mathtt{return}\}; \mathit{Sched}, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{reset}\ \{\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ \mathtt{true}\ k\}; [\![S]\!]\}; \mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, \\
&\qquad [\overline{x} \mapsto \overline{v}]; \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{R}\ \{\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ \mathtt{true}\ k\}; [\![S]\!]\}; \mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, \\
&\qquad [\overline{x} \mapsto \overline{v}]\dagger; \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{R}\ \{T := \mathtt{add}\ T\ \mathtt{true}\ k\}; \mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, \\
&\qquad [\overline{x} \mapsto \overline{v}, k \mapsto [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger]\dagger; \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{R}\ \{\mathtt{skip}\}; \mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, [\overline{x} \mapsto \overline{v}, k \mapsto [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger]\dagger; \rho; \Pi\dagger; \emptyset\dagger, \\
&\qquad \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, [\overline{x} \mapsto \overline{v}, k \mapsto [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger]; \rho; \Pi\dagger; \emptyset\dagger, \\
&\qquad \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{skip}\}; \mathtt{return}\}; \mathit{Sched}, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{return}\}; \mathit{Sched}, \rho'; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{skip}\}; \mathit{Sched}, \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathit{Sched}, \emptyset, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle
\end{aligned}
$$

The configuration we obtain from evaluation is exactly equal to the compiled configuration, thus for this case our claim holds.

– Case
$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S, \Pi\rangle \parallel \Theta \rightarrow n', \sigma' \triangleright n\langle e, S', \Pi'\rangle \parallel \Theta'$$
There are only two possible rules: S-Seq-Grd and S-Await. In both cases, it must be that $\sigma = \sigma'$, $\Pi = \Pi'$, $\Theta \equiv \Theta'$ and there exists some K such that $S = K[\mathtt{await}\ e]$ and $S' = K[\mathtt{skip}]$. Therefore, taking into account Lemma 1, the source and target configurations are compiled to:

$$
\begin{aligned}
[\![\mathit{cfg}_S]\!] &= \langle \mathtt{R}\ \{\mathtt{R}\ \{[\![K]\!][[\![\mathtt{await}\ e]\!]]\}; \mathtt{return}\}; \mathit{Sched}, \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&= \langle \mathtt{R}\ \{\mathtt{R}\ \{[\![K]\!][\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ [\![e]\!]\ k\}; \mathtt{skip}]\}; \mathtt{return}\}; \mathit{Sched}, \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
[\![\mathit{cfg}'_S]\!] &= \langle \mathit{Sched}, \emptyset, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{([\![e]\!], \mathtt{R}\ \{[\![K]\!][\mathtt{skip}]\}, \Pi\dagger)\}\rangle
\end{aligned}
$$

An example of this reduction can be seen in Figure 13.

– Case
$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S, \Pi\rangle \parallel \Theta \rightarrow n', \sigma' \triangleright n\langle S', \Pi'\rangle \parallel \Theta'$$
Rules matching this pattern are S-Seq-Fin, S-Seq-Step, S-If-True, S-If-False, S-While-True, S-Call. In the case of S-Seq-Step, we know that $S = S_0; S_1$ and $S' = S'_0; S_1$. By induction hypothesis, we get that
$$[\![\mathrm{env}_F]\!] \vdash [\![n, \sigma \triangleright n\langle S_0, \Pi\rangle \parallel \Theta]\!] \rightarrow [\![n', \sigma' \triangleright n\langle S'_0, \Pi'\rangle \parallel \Theta']\!]$$
As by the definition of the compilation function $[\![S]\!] = [\![S_0]\!]; [\![S_1]\!]$ and $[\![S']\!] = [\![S'_0]\!]; [\![S_1]\!]$, we obtain the needed result:
$$[\![\mathrm{env}_F]\!] \vdash [\![n, \sigma \triangleright n\langle S_0; S_1, \Pi\rangle \parallel \Theta]\!] \rightarrow [\![n', \sigma' \triangleright n\langle S'_0; S_1, \Pi'\rangle \parallel \Theta']\!]$$
For S-Seq-Fin, we know that $S = S_0; S_1$ and $S' = S_1$. Then the case follows by analyzing the step taken to reduce S_0.
The other cases are straightforward.

– One of the following three:

$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \Pi' \rangle \parallel n'\langle e, S, \Pi \rangle \parallel \Theta \rightarrow n', \sigma \triangleright n\langle \Pi' \rangle \parallel n'\langle S, \Pi \rangle \parallel \Theta$$
$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle e', S', \Pi' \rangle \parallel n'\langle e, S, \Pi \rangle \parallel \Theta \rightarrow n', \sigma \triangleright n\langle e', S', \Pi' \rangle \parallel n'\langle S, \Pi \rangle \parallel \Theta$$
$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle e, S, \Pi \rangle \parallel \Theta \rightarrow n, \sigma \triangleright n\langle S, \Pi \rangle \parallel \Theta$$

These three patterns match each of the scheduling rules. We will look only at the first one.

$$\frac{(\rho, \sigma) \models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \Pi' \rangle \parallel n'\langle e, S, \rho; \Pi \rangle \rightarrow n', \sigma \triangleright n'\langle S, \rho; \Pi \rangle \parallel n\langle \Pi' \rangle} \text{S-SCHED-FIN}$$

$$[\![cfg_S]\!] = \langle Sched, \emptyset, \sigma[T \mapsto \{([\![e]\!], [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, \rho; \Pi\dagger])\}]\rangle$$
$$[\![cfg'_S]\!] = \langle \mathtt{R}\ \{\mathtt{R}\ \{[\![S]\!]\}; \mathtt{return}\}; Sched, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto \emptyset]\rangle$$

The initial configuration will reduce as (with some of the steps omitted):

$$\begin{aligned}
&[\![\mathrm{env}_F]\!] \vdash \langle \mathtt{While}(\neg\mathtt{isEmpty}\ T)\mathtt{doreset}\ \{k := \mathtt{get}\ T; T := \mathtt{del}\ T\ k; \mathtt{invoke}\ k\},\\
&\qquad \emptyset, \sigma[T \mapsto \{([\![e]\!], [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, \rho; \Pi\dagger])\}]\rangle\\
&\mapsto \langle \mathtt{reset}\ \{k := \mathtt{get}\ T; T := \mathtt{del}\ T\ k; \mathtt{invoke}\ k\}; Sched,\\
&\qquad \emptyset, \sigma[T \mapsto \{([\![e]\!], [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, \rho; \Pi\dagger])\}]\rangle\\
&\mapsto \langle \mathtt{R}\ \{k := \mathtt{get}\ T; T := \mathtt{del}\ T\ k; \mathtt{invoke}\ k\}; Sched, \emptyset\dagger, \sigma[T \mapsto \{([\![e]\!], [\mathtt{skip}; \mathtt{R}\ \{[\![S]\!]\}, \rho; \Pi\dagger])\}]\rangle\\
&\mapsto^* \langle \mathtt{R}\ \{\mathtt{invoke}\ k\}; , [k \mapsto [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, \rho; \Pi\dagger]]\dagger, \sigma[T \mapsto \emptyset]\rangle\\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}; \mathtt{return}\}; Sched, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto \emptyset]\rangle\\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{[\![S]\!]\}; \mathtt{return}\}; Sched, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto \emptyset]\rangle
\end{aligned}$$

6 Conclusion

In this paper, we formalized a compilation scheme for cooperative multi-tasking into delimited continuations. For the source language, we extend While with procedure calls and operations for blocking and creation of new tasks. The target language extends While with shift/reset—the target language is sequential. We then proved that the compilation scheme is correct: reductions in the source language are simulated by corresponding reductions in the target language. We have implemented this compilation scheme in our compiler from ABS to Scala. The compiler covers a much richer language than our source language, including object-oriented features, and employs the experimental continuations plugin for Scala. The compiler is integrated into the wider ABS Tool Suite, available at http://tools.hats-project.eu/. We are currently formalizing the results of the paper in the proof assistant Agda.

Acknowledgements. This research was supported by the EU FP7 ICT project no. 231620 (HATS), the Estonian Centre of Excellence in Computer Science, EXCS, financed mainly by the European Regional Development Fund, ERDF, the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12, and the Estonian Science Foundation grant no. 9398.

References

1. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: ATEC 2002: Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference, pp. 289–302. USENIX (2002)
2. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
3. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
4. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. ACM SIGPLAN Notices - POPL 2004 39(1), 123–134 (2004)
5. Danvy, O., Filinski, A.: Abstracting control. In: LFP 1990: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp. 151–160. ACM (1990)
6. Haynes, C.T., Friedman, D.P., Wand, M.: Continuations and coroutines. In: LFP 1984: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, pp. 293–298. ACM (1984)
7. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
8. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science 365(1-2), 23–66 (2006)
9. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: PPPJ 1909: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, pp. 11–20. ACM (2009)
10. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
11. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
12. Wand, M.: Continuation-based multiprocessing. In: LFP 1980: Proceedings of the 1980 ACM Conference on LISP and Functional Programming, pp. 19–28. ACM (1980)

Extending UPPAAL for the Modeling and Verification of Dynamic Real-Time Systems

Abdeldjalil Boudjadar[1], Frits Vaandrager[2], Jean-Paul Bodeveix[3], and Mamoun Filali[3]

[1] CISS, Aalborg University, Aalborg, Denmark
[2] ICIS, Radboud University Nijmegen, Nijmegen, The Netherlands
[3] IRIT, Université de Toulouse, Toulouse, France

Abstract. Dynamic real-time systems, where the number of processes is not constant and new processes can be created on the fly like in object-based systems and ad-hoc networks, are still lacking a formal framework enabling their verification. Different toolboxes like Uppaal [21], Tina [10], Red [28] and Kronos [29] have been designed to deal with the modeling and analysis of real-time systems. Nevertheless, a shortcoming of these tools is that they can only describe static topologies. Other tools like Spin [18] allow the dynamic creation of processes, but do not consider time aspects. This paper presents a formal framework for modeling and verifying dynamic real-time systems. We introduce *callable timed automata* as a simple but powerful extension of standard timed automata in which processes may call each other. We show that the semantics of each call event can be interpreted either as an activation of the existing instance of the corresponding automaton (static instantiation), or a creation of a new concurrent instance (dynamic instantiation). We explore both semantical interpretations, static and dynamic, and give for each one the motivation and benefits with illustrating examples. Finally, we report on experiments with a prototype tool, which translates (a subset of) callable timed automata to UPPAAL systems.

Keywords: Dynamic real-time systems, timed automata, callable timed automata

1 Introduction

Timed automata (TA) [1] have been proposed as a powerful model for both timed and concurrent systems modelling. However, a dynamic framework for timed automata instantiation and applicability, to model dynamic system topologies like object-based systems and ad-hoc networks in which processes are created and triggered on the fly, is still lacking. Moreover, the modelling of timed automata as functional values, whereby a timed automaton can be called and applied to given parameters to generate outputs, instead of an independent component making computations and updating the system control is not explored. UPPAAL [6] is an integrated tool environment for editing, simulating and model checking

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 111–132, 2013.
DOI: 10.1007/978-3-642-40213-5_8,
© IFIP International Federation for Information Processing 2013

real-time systems modeled as networks of timed automata. The tool has been used successfully and routinely for many industrial case studies. Nevertheless, a shortcoming of UPPAAL is that it can only describe static network topologies, and does not incorporate a notion of dynamic process creation.

Unlike UPPAAL's *C-function* actions performing local sequential computations, this study consists of encoding the call mechanism into interacting processes, whereby communication on shared variables and synchronization with the external environment are enabled. The modelling of a timed automaton as a callable function which performs communications and interactions with the external environment enables it to be callable and triggerable by any other automaton. We introduce *callable timed automata* (CTA) as a formal framework for the modelling and analysis of dynamic timed systems, where the number of components (processes) may vary. The concept of callable timed automata enables, for a set of processes, to model a common behavior as an automaton callable by any other process originally performing such a behavior.

Syntactically, a callable timed automaton is a finite timed automaton [4] parameterized by a set of data, and triggered through the execution of a calling transition from another automaton. Moreover, a callable automaton may return results to its calling component. Semantically, we interpret this syntactical extension in different ways by considering different criteria like (1) *concurrency*: the activation of a callable process may be blocking for the corresponding calling process, wherein the former cannot progress while the callee one is running. Will both calling and callee components progress concurrently? (2) *instantiation*: the UPPAAL template's instantiation is static. Will the instantiation of callable TA be static (a constant number of instances initially created) or dynamic (for each call, a new instance is created on the fly)?

The ultimate goal of this paper is to provide a new formal framework for the modelling and verification of dynamic timed systems, where the number of processes is not constant, in terms of timed automata. To this end, we introduce an extension for structuring UPPAAL systems by integrating callable timed automata.

The rest of the paper is organized as follows. In Section 2, we cite existing related work. Section 3 motivates our proposal through a set of examples. In Section 4, we define callable timed automata and give their translation to UPPAAL TA. In Section 5, we review timed transition systems as a semantic basis. In Section 6, we define the semantics of both static and dynamic instantiations of CTA. Section 7 shows the implementation of CTA in UPPAAL. Section 8 presents the conclusion.

2 Related Work

In the literature, several frameworks [5,12,15,22,23,25,26] have been proposed to generalize the operational model of functions to a model of concurrent processes. Most of these proposals work on the encoding of the functional computation model λ-Calculus into the concurrent computation model π-Calculus. In [22],

Milner showed that λ-Calculus could be precisely encoded into π-Calculus. The SPIN tool [18] enables the verification of dynamic systems where concurrent processes can be created on the fly. Both creating and created processes progress together. The creation of a new process does not block the creating component execution i.e., a return is not needed to unlock the creating component. Similarly, the ADA language [13] enables the creation of tasks on the fly. After the creation of each task, the calling process waits until the new process is elaborated. Each process may perform a return immediately to unlock its calling component via action *accept*, or executes some actions then performs a return via statement *accept do* (RPC-like protocol[1]). Recently, there has been an amount of work focusing on recursive extensions of timed automata. Without considering synchronization, the authors of [27] define a restricted notion of recursive timed automata where their decidability results impose strong limitations on the number of clocks (at most 2 clocks). Moreover, either all clocks are passed by reference or none is passed by reference.

In our proposal, we introduce callable timed automata whereby we extend UPPAAL timed automata transition actions to concurrent process creation. Callable timed automata are referenced like functions and may interact with their environment. The semantics of each call event can be interpreted either as the activation of an existing instance of the corresponding template, or by the creation of a new concurrent instance of the callee automaton.

3 Callable Timed Automata

In this section, we introduce an extension of timed automata named *callable automata* where automata call each other. Unlike functions which are local computations getting their inputs as parameters before being triggered, a callable timed automaton is an open process which can interact with its external environment at anytime by accepting inputs, producing outputs and updating the system state. Syntactically, callable timed automata (CTA) are an extension of finite automata where transitions can be equipped by either a particular event `call`, to trigger the execution of another automaton, or again a `return` event to yield results. The call of a callable timed automaton can be parameterized by a set of expressions. Both call and return actions are used as a synchronization event instead of an update action. The execution of `call T` corresponds to the activation of an instance of template `T`. Obviously, the activation of an instance is preceded by its creation which can be performed either when the system starts or on the fly, i.e. when an automaton calls another one, it induces both instantiation and activation of the corresponding template.

In the semantical interpretations of call events, we may distinguish static and dynamic instantiations of callable timed automata. In fact, the interpretation of each call event depends on the nature of the callee template. To distinguish

[1] RPC is an acronym for *Remote Procedure Call*. It states the activation of a process (server) by another (client) such that the client process cannot progress while the server process does not perform a return.

between static and dynamic interpretations, we associate to each CTA signature either a finite number n or an infinite one ∞. Namely, if the template signature states a finite number n of instances, then each call event for that template is considered to be static. Otherwise, in the case of ∞, the call event will be considered to be dynamic.

3.1 Static Instantiation

In this subsection, we consider the situation that each callable timed automaton is instantiable through a constant number of instances, that may be initially created when the system starts. The execution of each call event corresponds to the activation of an instance of the callee template, which may delay and interleave with the execution of other components. That is the same case as for UPPAAL-Port [17] where components trigger each others. Each of the instances will be reinitialized for each activation (call) with the corresponding parameters. In fact, the callable automaton instances are considered as any other instance associated to a normal UPPAAL template. Moreover, with such an interpretation, a callable automaton T can be called concurrently in the limits of its number of instances $\mathcal{I}(T)$.

Formally, the call event is blocking where the calling component cannot run any other transition while its callee automaton has not performed a return. Likewise, a CTA may block call events from components other than the current callers if free instances are not available. The calling component gets the control back when the execution of the callee instance emits a return event. The return event of a callee instance does not state its termination. The execution of a callee instance can be atomic, which agrees with the UPPAAL action semantics.

The static instantiation applicability of callable timed automata covers a large spectrum of the RPC-based systems. An example of such an instantiation can be found in UPPAAL-Port, where a system is structured as a set of hierarchical components executed in a sequence. When the execution of a component has completed, it triggers the (non-atomic) execution of another component by activating its trigger-ports. Without considering hierarchy, one can distinguish that an UPPAAL-port system can be translated to a set of callable automata in a systematic way. Such a translation consists of replacing the activation of trigger-ports of each component by a call made by the last transition of its triggering component.

3.2 Example 1 (Static Instantiation)

We reuse the UPPAAL expression of the well known *Train-Gate* example [6], depicted in Figure 1. In fact, such an example models the train crossing concurrency, where a set of trains request concurrently access to a unique crossing point, *the critical section*, in order to continue on their respective routes. The crossing point is governed by a gate which each train must signal to gain crossing authorization.

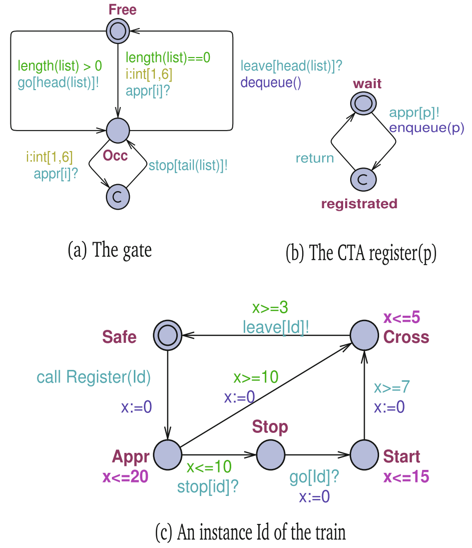


Fig. 1. The *Train-Gate* Example

In order to distinguish between train instances, each one has a unique identifier Id. As the access request is the same for all trains, we model this common behavior (access request) by a new parameterized callable timed automaton named $Register$, and by that trains get rid of requesting their own access authorization. The automaton $Register$ can be called by any train intending to cross the gate.

When a train Id approaches the crossing point, it calls the automaton $Register$ with its own identifier Id. The automaton $Register$ notifies the $Gate$, which the train Id is approaching, through a synchronization on channel $appr$, and inserts Id into the waiting list $list$. Whenever the execution of automaton $Register$ is over for a given call by reaching the return action, the corresponding calling train can resume. Depending on the availability of the $Gate$, such a train (Id) crosses immediately or stops for a delay specified by a constraint on clock x, waiting to be on the front of $list$ then crosses the gate. Accordingly, the automaton $Register$ becomes available for accepting other calls by any train intending to cross the gate.

3.3 Dynamic Instantiation

In this interpretation, a varying number of instances can be dynamically associated to each callable automaton: each call event corresponds to the creation of a new instance of the callee automaton. Template instances are created on the fly through the execution of the corresponding calls. Each newly created instance will be simultaneously triggered. Hence, the call event is not blocking for other calling components. Moreover, both calling and callee instances may progress concurrently, after performing a return. In fact, in the dynamic instantiation the return event of an instance enables to yield its results but does not state its termination. i.e. an instance may run other transitions after performing a return. The termination of an instance execution is stated by reaching a final location. The dynamic instantiation of callable timed automata leads to building the structure of the system on the fly: the system has different numbers of instances on different executions and at different dates.

The dynamic feature of such an instantiation is suitable to model object-based systems, ad-hoc networks, fault tolerant and DataBase Management systems (DBMS) where components (objects, hosts, processes) are created on the

fly. For example, in the case of DataBase Management systems, when the execution of a process requires to read data from a database, it calls the Reader module of DBMS by creating an instance of the former to fetch data.

3.4 Example 2 (Dynamic Instantiation)

The sieve of Eratosthenes is a simple algorithm for finding all prime numbers up to a given integer M. Given a list of numbers, the algorithm iteratively marks as a non-prime the multiples of each prime, starting with the multiples of 2. It runs across the table until the only numbers left are prime.

As depicted in Figure 2, we have implemented this algorithm by the parallel composition of 2 automata: `main` and `element`. In fact, we model the table elements by the automaton so-called `element`. Each instance of template `element` is parameterized by a natural number (1 of template `main`) which states its identifier, and another integer number (2 of template `main`) to retrieve its prime number. Moreover, each instance has 2 local variables: *self* to store its identifier (parameter), and *myprime* to store the value of the corresponding prime number (parameter). To allow the communication of instances, we declare a vector *next* of M channels.

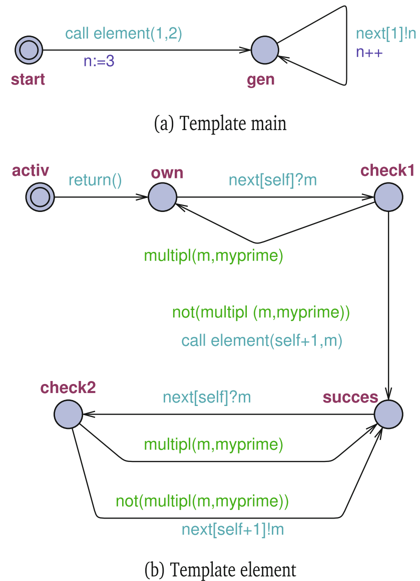


Fig. 2. The sieve of Eratosthenes

The system is managed by another automaton so-called `main`, which creates the first instance of automaton `element`. Such an instance gets as effective parameters the identifier of the first instance (1), and the corresponding prime number (2). After that, automaton `main` increments iteratively the number n to be checked and sends it to that instance (of template `element`) through channel $next[1]$ [2].

Once the system is triggered, the automaton `main` moves from location *start* to location *gen* (generate) by executing the call action *call element*$(1, 2)$, and updating n to 3. Such a call creates the first instance of template `element`, which is identifiable by $self = 1$ and $myprime = 2$. This instance performs a return to unlock its caller and moves to its location *own*. The automaton `main` sends the first value n to be checked to the newly created instance, of template `element`, on channel *next[1]*. Through the reception of the first message $next[self]?m$, the

[2] In fact, channels *next* are parameterized by the number to be checked. We may consider shared variables to implement the data communication over channels.

current `element` instance checks whether or not the received value of m is a multiple of its own prime number *myprime*.

If m is a multiple of *myprime* then the received value of m will be ignored, and the current instance of `element` moves back from location *check1* to location *own*. Otherwise, the current instance of template `element` requests the creation of another instance, through the statement *call* $element(self + 1, m)$, and moves to location *succ* (successor). At this level, the first instance of `element` is waiting for the reception of another number to be checked, sent by `main`. On a reception $next[self]?m$ of a new value which is not a multiple of *myprime*, the instance of `element` sends that value to its successor instance via channel $next[self + 1]$, which corresponds in this case to $next[2]$. Similarly, each new instance of `element` behaves in the same way as the first one. Herein, one can distinguish that each new number, sent by automaton `main`, crosses a sequence of `element` instances until it is dropped, the case of a multiple of a discovered *myprime*, or registered as a new prime number with the creation of a new instance of template `element`.

4 Timed Automata Extension

The modeling and verification of real-time systems, via timed automata, are mature topics to which a large amount of work has been devoted during the last two decades. However, the modeling and verification of dynamic real-time systems, where the topology (global architecture and number of components) may change during the execution of systems, constitute a perspective and an active field of research.

In this section, we give the formal basis of callable timed automata (CTA) where transition actions can be internal, external, a `call` of another callable timed automaton, or again a `return`. Then, we show how callable timed automata can be translated to UPPAAL ones, and establish an important result stating that the semantics of CTA and that of their translation to UPPAAL timed automata are bisimilar (Figure 3). In fact, the translation enables us to reuse the UPPAAL toolbox for the verification of dynamic timed systems modeled with CTA. Let us introduce the following notation.

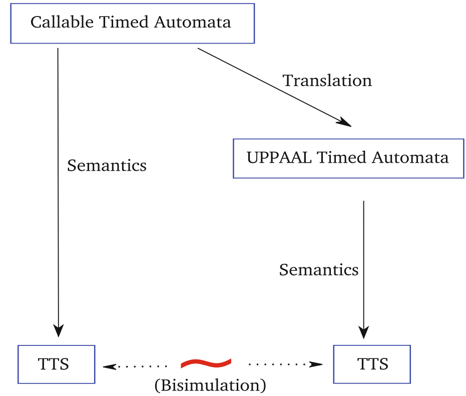


Fig. 3. Semantics and translation of CTA

Notation. We assume a universe $\mathcal{V}$ of variables. To each variable $v \in \mathcal{V}$ we associate a nonempty set of values, referred to as the type of v and denoted $type(v)$. Moreover, we associate to each variable $v \in \mathcal{V}$ a default initial value

$d_v^0 \in type(v)$. A variable v whose type equals the set $\mathbb{R}_{\geq 0}$ of non-negative real-numbers is called a *clock*. We assume that the default initial value of all clocks equals 0. Let $V \subseteq \mathcal{V}$ be a set of variables.

- A valuation of V is a function that maps each variable to an element of its type. We use $Val(V)$ to denote the set of valuations of V.
- $\mathfrak{E}(V)$ defines the set of expressions built over V. To each expression $e \in \mathfrak{E}(V)$ we assign a type $type(e)$. Each expression induces a state transformer, that is, $[\![e]\!] : Val(V) \rightarrow Val(V)$. We call an expression side effect free if $[\![e]\!]$ is the identity function. Each expression also denotes a value for any valuation: $\langle\langle e \rangle\rangle : Val(V) \rightarrow type(e)$.
- $\mathfrak{P}(V)$ defines the set of predicates built over V. If ϕ is a predicate over V then $[\![\phi]\!] : Val(V) \rightarrow Bool$ gives the truth value of ϕ for any given valuation of V.
- For a function f defined on a domain $dom(f)$, we write $f\lceil X$ the restriction [8] of f to X, that is the function g with $dom(g) = dom(f) \cap X$ such that $g(z) = f(z)$ for each $z \in dom(g)$.
- Two functions f and g are compatible [8], denoted $f \heartsuit g$, if they agree on the intersection of their domains, that is, $f(z) = g(z)$ for all $z \in dom(f) \cap dom(g)$.
- We denote by $f \triangleright g$ the left overriding function defined on $dom(f \triangleright g) = dom(f) \cup dom(g)$ where f overrides g for all elements in the intersection of their domains. For all $z \in dom(f \triangleright g)$,

$$(f \triangleright g)(z) \triangleq \begin{cases} f(z) \text{ if } z \in dom(f) \\ g(z) \text{ if } z \in dom(g) - dom(f) \end{cases}$$

  Similarly, we define the dual right overriding operator by $f \triangleleft g \triangleq g \triangleright f$.
- We define $f \parallel g \triangleq f \triangleright g$ when f and g are compatible.

4.1 UPPAAL Timed Automata

UPPAAL is an integrated tool environment for editing, simulating and model checking real-time systems modeled as networks of timed automata. The tool has been used successfully and routinely for many industrial case studies. Nevertheless, a shortcoming of UPPAAL is that it can only describe static network topologies, and does not incorporate a notion of dynamic process creation. Moreover, UPPAAL does not incorporate a notion of one automaton calling another, like a function, even though this last concept can be encoded within UPPAAL using a pair of handshakes.

In fact, UPPAAL timed automata [6] are extensions of the classical ones [1] where one level hierarchy of local/global variables, committed locations, communication and priorities have been introduced. Besides, in the UPPAAL language timed automata are defined within a global common context.

Definition 1. (Global context) *A global context $\mathcal{C} = \langle \Sigma, V^g, Init^g, C \rangle$ consists of a finite set of automata names $\Sigma \subseteq \mathcal{T}$, a finite set of global variables*

$V^g \subseteq \mathcal{V}$, the initial valuation $Init^g$ of global variables V^g and a finite set of channels C.

Throughout this paper we do not distinguish between clock and normal variables. Each variable of $\mathcal{V}$ is either a clock or a normal variable. By now, we give the structure of a timed automaton defined on a global context.

Definition 2. (Timed automaton) *Given a global context $\mathcal{C}$, a timed automaton (TA) is a tuple $\langle Q, q^0, K, V^l, Init^l, Inv, \rightarrow\rangle$ where Q is the set of locations, $q^0 \in Q$ is the initial location, V^l is the set of local variables, $Init^l$ is the initial valuation of local variables, $Inv : Q \rightarrow \mathfrak{P}(V)$ associates an invariant to each location, $K \subseteq Q$ is a set of committed locations, and $\rightarrow \subseteq Q \times \mathfrak{P}(V) \times \Lambda \times \mathfrak{E}(V) \times Q$ is the transition relation, where $V = V^l \cup V^g$ and $\Lambda = C? \cup C! \cup \{\tau\}$.*

For the sake of simplicity, we write $q \xrightarrow{G/\lambda/a} q'$ for $(q, G, \lambda, a, q') \in\rightarrow$. The composition of timed automata, so-called networks of timed automata (NTA), enables to model a system as a flat set of interconnected components. Each component (TA) interacts with its external environment through communication on shared variables and synchronization of actions.

In a variant of UPPAAL called UPPAAL-Port [17], hierarchical compositions are enabled whereby the system can be modeled as a set of components. Each component may encapsulate other components. Several proposals [6,8,11,14] studying the composition of UPPAAL timed automata have analyzed their properties. The authors of [6] define a non compositional semantics of UPPAAL NTA. In [8,11], the authors define a compositional semantics of NTA and establish some properties like the preservation of system invariants. In [14,20], the semantics of TA composition is not compositional because the product of TA semantics is not associative. Counter-examples are given in [7,9].

4.2 Callable Timed Automata

Callable timed automata provide a formal framework for the modelling and analysis of dynamic timed systems. In fact, the concept of callable timed automata enables, for a set of processes, to model a common behavior as an automaton callable by any other process originally performing such a behavior.

Unlike UPPAAL callable C-functions, a callable timed automaton can interact with the other components and call other callable automata. However, in the case of static instantiation, in order to avoid deadlock due to mutually dependent executions, a callable timed automaton cannot call its own hierarchical calling components. In fact, for the static interpretation, the calling component cannot progress while its current callee component is running. Once the callee TA execution is over, the corresponding calling component may resume the control and continue its execution. However, for the dynamic instantiation, after performing a return to unlock its calling component, a callee component may progress together with the execution of its calling component. Thus, in the static instantiation, the *return* action represents the end of the call execution of callable TA

whereas, in the dynamic instantiation, it is considered as an ordinary action. Obviously, a system of CTA must contain at least one triggering TA (root) to activate CTA.

We assume a universe $\mathcal{T}$ of automata names, and associate to each automaton name $T \in \mathcal{T}$ a return type $\mathcal{R}(T)$, the number of instances to be created $\mathcal{I} : \mathcal{T} \rightarrow \mathbb{N}_{>0} \cup \{\infty\}$ and a formal parameter $p_T \in \mathcal{V}$. In fact, the maximal number of instances to be created for each template is either a strictly positive number (> 0) if the template is statically instantiable, or an infinity (∞) in the case of dynamic instantiation.

In this paper, we only consider automata with a single formal parameter. Automata with multiple parameters may be encoded using variables of type vector, record or union in the same way as simple types and without affecting our framework.

We introduce expressions of type automaton and write $\mathfrak{E}(\mathcal{T}, \mathcal{V})$ for the set of expressions $\{T(e) \mid T(p_T) \in \Sigma \wedge e \in \mathfrak{E}(\mathcal{V}) \wedge type(e) = type(p_T) \wedge e$ is side effect free$\}$. Formally, a callable timed automaton is given by:

Definition 3. (Callable timed automaton) *Let $\mathcal{C} = \langle \Sigma, V^g, Init^g, C \rangle$ be a global context. A callable timed automaton (CTA) for $\mathcal{C}$ is a tuple $\langle T, Q, q^0, F, V^l, Init, Inv, \rightarrow \rangle$ where Q, q^0, $Init^l$ and Inv are the same as for TAs:*

- *$T \in \Sigma$ is the automaton name,*
- *$F \subseteq Q$ a set of final locations,*
- *$V^l \subseteq \mathcal{V}$ is a set of local variables; we require $V^g \cap V^l = \emptyset$, $p_T \in V^l$, and write $V = V^g \cup V^l$,*
- *$\rightarrow \subseteq Q \times \mathfrak{P}(V) \times \Lambda \times \mathfrak{E}(V) \times Q$ is the transition relation which, for each transition, consists of a source location, a guard, a label, an action and a target location. Here $\Lambda = C? \cup C! \cup \{\tau\} \cup (V \times \Sigma \times \mathfrak{E}(V)) \cup \mathfrak{E}(V)$ is the set of transition labels. Each transition label can be a synchronization on a channel, an internal event, a call of another automaton, or a return action.*

We write $q \xrightarrow{G/\lambda/a} q'$ for $(q, G, \lambda, a, q') \in \rightarrow$. Moreover, if $\lambda = (x, T', e) \in V \times \Sigma \times \mathfrak{E}(V)$ then we refer to the transition as a call transition and write $q \xrightarrow{G/x:=call\ T'(e)/a} q'$. In this case, we require that $type(e) = type(p_{T'})$ and $type(x) = \mathcal{R}(T')$. Similarly, if $\lambda = e \in \mathfrak{E}(V)$ then we refer to the transition as a return transition and use the notation $q \xrightarrow{G/return(e)/a} q'$. In this case we require that $type(e) = \mathcal{R}(T)$. Intuitively, via a *call* $T'(e)$-transition automaton T calls automaton T' with a parameter value that can be obtained by evaluating expression e. A *return*(e)-transition is used to return the value of expression e. If the return type of an automaton T is void, we use *return*() and just keep *call* $T(E)$ to call the automaton T, omitting the assignment "$x :=$". Furthermore, callable timed automata should satisfy the following wellformedness conditions: final locations do not have outgoing transitions, and return actions are side effect free. We call *subprogram* a CTA of which each return transition leads to a final location. Moreover, we associate to each automaton name a CTA template (record): $\mathcal{D} : \mathcal{T} \rightarrow CTA$.

4.3 Translation of Callable TA to UPPAAL TA

In order to reuse the UPPAAL toolbox, we translate callable timed automata to UPPAAL TA. Hence, as stated in the previous section, to make the translation and implementation of CTA easier the user provides the nature of each template instantiation. In fact, through $\mathcal{I}(T)$ the user states whether the template T is instantiable statically or dynamically. Moreover, the user specifies the number of instances to be created, for each template, in the case of static instantiation. Since calling and callee components may not access each others local variables, we consider the UPPAAL communication through shared variables.

As shown in Figure 4, for translating the calling transition $q \xrightarrow{x<=0/y:=call\ T(e)/} q'$, the expression e is assigned to a new shared variable $param$ [3]. Thereafter, the value of such a shared variable will be copied into the local variable $p_T \in V^l$ of the callee automaton $(T(p_T) \in \Sigma)$, as depicted in the bottom of Figure 5.

Mainly, the translation consists of splitting each calling transition of CTA into two synchronizing transitions, as shown in Figure 4. The first transition is an output on a particular channel cal, to activate the corresponding callee CTA, which engages with the assignment of expression e to shared variable $param$, whereas the second transition is an input on a particular channel ret, with the assignment of value $result$ to variable y requesting the call. The execution of the former transition states the termination of the call execution. Both transitions, resulting from the translation of a call, are linked through a new intermediate location q_{int} relative to each pair (y, t), where t is the original calling transition and y is the variable requesting the call. In fact, we use the notation $t : q \xrightarrow{G/\lambda/a} q'$ to state that t is the current transition name, which will be used to reference this transition.

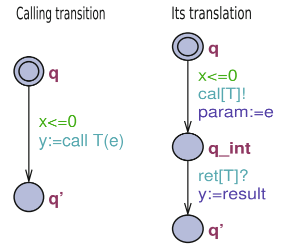


Fig. 4. The translation of calls

In Figure 5, we show how the structure of a callable automaton (top) can be translated to that of an UPPAAL one (bottom). The translation consists of adding a new initial location q_{init} as the triggering point (activation) of the corresponding UPPAAL TA. This location will be linked to the original initial location q^0 of CTA through an input synchronizing transition on channel $cal[T]$, engaging with the assignment of shared variable $param$ to the CTA local variable p, dedicated to receive the parameter value.

When it meets a return event, the callee CTA yields its result to its calling through a synchronizing transition on channel $ret[T]$, which assigns the result r to shared variable $result$ and unlocks the calling component. Moreover, all CTA final locations are linked to newly inserted location q_{init} via an empty committed transition in order to make CTA available for other calls.

[3] In fact, the type $type(param)$, resp $type(result)$, is the union of all of the parameter, resp return, types used in the model.

Remark. We have associated to each callable timed automaton T a pair of channels $(cal[T], ret[T])$. Such channels can be used by any other automaton T' intending to call automaton T. Moreover, the set of parameters $\{param\}$, respectively $\{result\}$, depends on the number and types of call, respectively return, parameters. Such variables are re-used for the whole model because the synchronizations on cal and ret channels are atomic transitions.

Definition 4. (TA corresponding to a CTA) *Given a CTA* $\langle T, Q, q^0, F, V^l, Init^l, Inv, \rightarrow_T \rangle$ *for a global context* $\mathcal{C} = \langle \Sigma, V^g, Init^g, C \rangle$ *with* $T(p_T) \in \Sigma$*, its translation to a TA is defined by* $\langle Q \cup Q^{int} \cup \{q_{init}\}, q_{init}, F, V^l, Init^l, Inv', \rightarrow \rangle$ *over the global context* $\langle \Sigma, V^g \cup \{param, result\}, Init^g, C \cup \{cal, ret\} \rangle$ *where* $Inv'(q) = Inv(q)$ *if* $q \in Q$ *else true and* $\rightarrow$ *is the smallest relation such that:*

$$\frac{q \xrightarrow{G/\lambda/a}_T q' \quad \lambda \in C! \cup C? \cup \{\tau\}}{q \xrightarrow{G/\lambda/a} q'} \; Action \qquad \frac{q \in F}{q \xrightarrow{\top/\tau/skip} q_{init}} \; Restart$$

$$\frac{}{q_{init} \xrightarrow{\top/cal[T]?/p_T:=param} q^0} \; Activate \qquad \frac{q \xrightarrow{G/return(e)/a}_T q'}{q \xrightarrow{G/ret[T]!/result:=e,a} q'} \; Return$$

$$\frac{t : q \xrightarrow{G/x:=call\ T'(e)/a}_T q'}{q \xrightarrow{G/cal[T']!/param:=e} q_t \xrightarrow{\top/ret[T']?/x:=result,a} q'} \; Call$$

where skip is an empty action (identity), $Q^{int} = \{q_t \mid t : q \xrightarrow{G/x:=call\ T'(e)/a} q'\}$ *is a set of intermediate locations introduced when splitting the calling transitions as shown in Figure 4, and* q_{init} *is the new initial location of the resulting TA, again illustrated in Figure 5.*

In fact, this definition translates a CTA and its global context to a timed automaton, where the final locations are marked committed in order to get instances immediately available after the end of each call. Each instance of the template T is processed in the same way. Therefore, the translation of a CTA is a network of timed automata defined on the translation of the global context $\mathcal{C}$ where to each instance of the CTA T corresponds a TA.

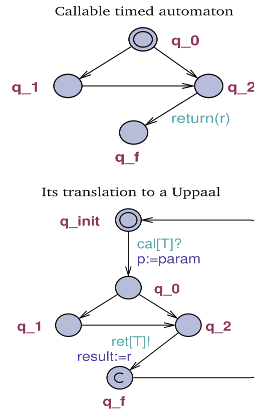


Fig. 5. TA of a CTA

Transition rule *Action* states that non calling transitions of the CTA are held without any change in the corresponding translation. Rule *Restart* enables the resulting TA to join its new initial location q_{init} from each final location. Via rule *Activate*, the execution of a callable TA translation is activated through an enabled (guard = $\top$) synchronizing transition. The former leads to reach the old initial location q^0 of the CTA, and updates the value of parameter p_T according to value of variable

param. Rule *Return* states that whenever a callable automaton emits a `return` event, its translation yields the results to its calling (parent) TA through a synchronization on channel *ret*, with the assignment of result e to shared variable *result*. Finally, rule *Call* is explained via Figure 4.

In the same way, the translation of a network of CTA, defined by a root CTA, a set of template definitions $\mathcal{D}$ and the maximal number of instances associated to each template $\mathcal{I}$ which is supposed to be **bounded**, is a NTA containing the translation of each CTA replicated according to their number of instances.

In the case of dynamic-instantiable CTA (infinite number of instances), for each CTA T we choose a finite number n_T of instances for each infinite number $\mathcal{I}(T)$, then we translate the new CTA model to UPPAAL. Thus, if the number of simultaneously active instances of each T is lower than the corresponding chosen number, the properties of the checked TA model are those of the original CTA model. In order to check that the chosen numbers are sufficient, we use the UPPAAL model-checker to prove that for each T there always exists an instance in its initial state.

Otherwise, we retry with higher values n_T, for each T whose the number of instances has been reached, and redo the checking process. However, such a process may not terminate. A perspective of this section is to provide a tool for inferring automatically the sufficient number of instances for each dynamic-instantiable CTA. Such a tool could be based on the decision procedure for the boundedness of Petri nets.

5 Semantical Model: TTSs

In order to ensure the translation correctness, we define the semantics of both UPPAAL timed automata and callable TA in terms of timed transition systems (TTS). We study then the bisimilarity between the CTA direct semantics and the translation-based one. In fact, we study the bisimilarity between the semantics of CTA composition and that of their translation, defined in a compositional way. To this end, we extend timed transition systems with local and global variables, and review their timed bisimulation relation and associative product, according to [8]. Moreover, we consider the static priority *Committedness*, which is useful to specify that certain behaviors need to be executed atomically, without interleaving of lower priority behaviors from other components. In general, the states of a TTS constitute a proper subset of the set of all valuations of the state variables. This feature is used to model the concept of location invariants in timed automata.

Definition 5. (TTS) *A Timed Transition System over a set of channels C is a tuple $\langle \mathcal{G}, \mathcal{L}, S, s^0, \rightarrow \rangle$ where $\mathcal{G}$ and $\mathcal{L}$ are respectively the sets of global and local variables, $S \subseteq Val(V)$ is the set of states with $V = \mathcal{G} \cup \mathcal{L}$, $s^0 \in S$ the initial state and $\rightarrow \in S \times (C! \cup C? \cup \{\tau\} \cup \Delta) \times \mathbb{B} \times S$ is the transition relation. Δ is the time domain and $\mathbb{B}$ states whether or not a transition is committed. A state s of a TTS is called committed, denoted $Comm(s)$, if it enables an outgoing committed transition $(s, l, \top, s')$.*

Furthermore, a TTS must satisfy a wellformedness condition : in a committed state neither time-passage steps nor uncommitted τ may occur. Thus, time transitions (with labels in Δ) are non committed.

In fact, the state space S will be used to encode the location invariants of timed automata. Here and elsewhere, we write $s \xrightarrow{\lambda,b} s'$ for a transition $\langle s, \lambda, b, s' \rangle \in\rightarrow$ linking the state s to another state s' through an event λ and having the committedness priority b. This former is considered to be false ($\bot$) if absent. Formally, the predicate $Comm$ is defined by:

$$Comm(s) = \begin{cases} \top & \text{If } \exists\ \lambda\ s' \mid s \xrightarrow{\lambda,\top} s' \\ \bot & \text{Otherwise} \end{cases}$$

Through location committedness, certain (lower-priority) behavior are ruled out which may lead to serious reductions in the state space of a model. By now, we define the simulation relation of TTSs [8]. In fact, such a relation is used to show whether a TTS implements another. The simulation relation can be established through the inclusion of traces where, from a common state, we check that each transition of the simulated system can be triggered in the simulating one.

Definition 6. (Timed step simulation) *Given two TTSs T_1 and T_2 having the same set of global variables, we say that a relation $R \subseteq S_1 \times S_2$ is a timed step simulation from T_1 to T_2, provided that $s_1^0 R s_2^0$ and if $s\ R\ r$ then*

- $s\lceil\mathcal{G}_1 = r\lceil\mathcal{G}_2$,
- $\forall u \in Val(\mathcal{G}_1) : s[u]R\ r[u]$,
- *if* $Comm(r)$ *then* $Comm(s)$,
- *If* $s \xrightarrow{\lambda,b} s'$ *then either there exists an* r' *such that* $r \xrightarrow{\lambda,b} r'$ *and* $s'Rr'$, *or* $\lambda = \tau$ *and* $s'Rr$.

where $s[u]$ states the update of state s according to valuation u. We write $T_1 \preceq T_2$ when there exists a timed step simulation from T_1 to T_2. In fact, such a definition maps each transition of T_1 to a transition of T_2 given that global variables have the same valuations. Accordingly, T_1 and T_2 are bisimilar if there exists a timed step simulation R from T_1 to T_2 such that R^{-1} is a timed step simulation from T_2 to T_1. In order to study the semantics of timed automata composition, we define the product of TTSs, according to [8], which is a partial operation that is only defined when TTSs initial states are compatible, i.e. $s_1^0 \heartsuit s_2^0$.

Definition 7. (Parallel composition of TTSs) *Given two TTSs T_1 and T_2 with $s_1^0 \heartsuit s_2^0$, their parallel composition $T_1 \parallel T_2$ is defined by the tuple $\langle \mathcal{G}, \mathcal{L}, S, s_i^0, \rightarrow \rangle$ where $\mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2$, $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$, $S = \{r \parallel s \mid r \in S_1 \wedge s \in S_2 \wedge r \heartsuit s\}$, $s^0 = s_1^0 \parallel s_2^0$ and $\rightarrow$ is the smallest relation such that:*

$$\frac{r \xrightarrow{\lambda,b}_i r'}{r \parallel s \xrightarrow{\lambda,b} r' \triangleright s}\ Ext \qquad \frac{r \xrightarrow{\tau,b}_i r' \quad Comm(s) \Rightarrow b}{r \parallel s \xrightarrow{\tau,b} r' \triangleright s}\ Tau$$

$$\frac{\begin{array}{c} r \xrightarrow{c!,b}_i r' \quad s[r'] \xrightarrow{c?,b'}_j s' \quad i \neq j \\ Comm(r) \vee Comm(s) \Rightarrow b \vee b' \end{array}}{r \parallel s \xrightarrow{\tau,b \vee b'} r' \triangleleft s'}\ Sync \qquad \frac{r \xrightarrow{\delta}_i r' \quad s \xrightarrow{\delta}_j s' \quad i \neq j}{r \parallel s \xrightarrow{\delta} r' \parallel s'}\ Time$$

$i,\ j$ range over $\{1,2\}$ and $b,\ b'$ range over $\mathbb{B}$. The set of variables of the product is simply obtained by the union of both component variables. Moreover, the states, respectively initial states, of the product are obtained by merging the states, respectively initial states, of individual TTSs. The notation $Comm(q) \Rightarrow b$ with $t : s \xrightarrow{\lambda,b}_i s'$ states that t must be committed if there exists another outgoing committed transition from s. Otherwise stated: a transition cannot be hidden by a lower-priority transition.

Rule *Ext* represents potential synchronizations that the TTS T_i may be willing to engage in with its environment. The committedness of such transitions is not checked because it may be that a compatible committed transition will synchronize with the current transition of T_i making then the resulting transition committed. Rule *Tau* induces an internal transition of the composition from an internal transition of a component T_i. Rule *Sync* describes the synchronization of components T_i and T_j on channels $c \in C$ if their labels are compatible, and the input transition is still triggerable according to the valuation associated to the output transition target state r'. The resulting transition, labelled by the internal event τ, is committed if at least one of the involved transitions (output, input) is committed. Hence, a non-committed synchronization may only occur if both components are in uncommitted states. Finally, rule $TIME$ states that a delay δ of the composition may occur when both components perform a delay δ.

Theorem 1. (Associativity) *Let T_1, T_2 and T_3 be TTSs with their initial states pairwise compatible, then $(T_1 \parallel T_2) \parallel T_3 = T_1 \parallel (T_2 \parallel T_3)$.*

In the following, we define the semantics of UPPAAL timed automata through TTS where committed transitions of TTS are those outgoing from TA committed locations.

Definition 8. (TTS semantics of a TA) *Given a global context $\mathcal{C} = \langle \Sigma, V^g, Init^g, C\rangle$, the TTS associated to a timed automaton $\langle Q, q^0, K, V^l, Init^l, Inv, \rightarrow_{ta}\rangle$ is defined by $\langle V^g, V^l \cup \{loc\}, S, s^0, \rightarrow\rangle$ where `loc` is a fresh variable with type Q, $W = V^g \cup V^l \cup \{loc\}$, $S = \{v \in Val(W) \mid v \models Inv(v(loc))\}$, $s^0 = Init^g \cup Init^l \cup \{loc \mapsto q^0\}$ and the transition relation is defined by:*

$$\frac{q \xrightarrow{G/\lambda/a}_{ta} q' \quad s(loc) = q \quad s \models G \quad b \Leftrightarrow (q \in K)}{s \xrightarrow{\lambda,b} a(s \triangleleft \{loc \mapsto q'\})}\ Act \qquad \frac{s(loc) \notin K}{s \xrightarrow{\delta,\bot} s \oplus \delta}\ Time$$

We have introduced a new local variable `loc` to state the TA current location. Each state of the TTS corresponds to a valuation of TA variables where the invariant of the corresponding location holds. Moreover, the TTS transitions are inferred from the transitions and locations of TA. In fact, rule *Act* states that to each TA transition, we associate a TTS transition if the current location `loc` corresponds to the source location q of TA transition, and the TTS current state s satisfies the guard G of the TA transition. Through rule $Time$, we associate to each non-committed location of TA a TTS non-committed transition. The former adds an amount δ to all clock variables. One may distinguish that $Time$ transitions do not update local states and non-clock variables.

6 Semantical Interpretations

By now, we define the semantics of callable timed automata instantiation in terms of TTS. In fact, such a semantics considers a callable automaton (template) together with its instances. Mainly, we distinguish two different instantiations: static and dynamic. In fact, the static instantiation corresponds to implement each callable template through a finite (constant) number of instances, may be initially created, whereas the dynamic instantiation of a callable automaton consists of creating a (possibly infinite) set of instances on the fly when executing the system. Each instantiation mechanism is suitable for a given kind of applications, whereby the modelling of systems becomes much more natural. Let us introduce the following elements:

- We extend the set of locations by introducing, for each calling transition t a new location $\bar{t}$. Such a location will be used to wait for a return of the call made over transition t.
- In order to distinguish between different instances of the same template, a fresh identifier Id is assigned to each instance.
- We introduce a new local variable **templ** such that, an instance Id is an instantiation of the template T if $Id.\mathbf{templ} = T$.
- We have also introduced a new local variable **ParId** in order to identify for whom (parent identifier) an instance (Id) performs a return. In fact, the variable $Id.\mathbf{ParId}$ stores the identifier of the current caller of Id.
- The local variables of instance Id are renamed by prefixing each one by the identifier Id.
- The notation $[\![e]\!]_s^{Id}$ states the valuation of expression e according to state s, where the template local variables occurring in e are replaced by the corresponding local variables of instance Id.

Definition 9. (CTA instantiation semantics) *Given a global context* $\mathcal{C} = \langle \Sigma, V^g, Init^g, C \rangle$*, the instantiation semantics of the callable timed automaton* $\langle T(p_T), Q, q^0, F, V^l, Init^l, Inv, \rightarrow_T \rangle$ *is defined by the TTS* $\langle V^g, Id.V^l \cup Id.$

$\{\mathbf{loc}, \mathbf{templ}, \mathbf{ParId}\}, S, s^0, \rightarrow\rangle$ [4] *over the set of channels* C *where* $Id = fresh(\emptyset)$ *is the identifier of the initial instance,* $S = \{s \in Val(W) \mid s \models Inv(s(\mathbf{loc}_T))\}$, $s^0 = Init^g \cup Init^l \cup \{Id.\mathbf{loc} \mapsto q^0\}$, $W = V^g \cup \bigcup_{i\in\mathcal{I}(T)}\{Id_i.V^l \cup \{Id_i.\mathbf{loc}, Id_i.\mathbf{templ}, Id_i.\mathbf{ParId}\}$ *and* $\rightarrow$ *is the smallest relation such that:*

$$\frac{q \xrightarrow{G/\lambda/a}_T q' \quad s(Id.\mathbf{templ}) = T \quad s(Id.\mathbf{loc}) = q \quad s \models G}{s \xrightarrow{\lambda,\bot} a_{Id}(s \lhd \{Id.\mathbf{loc} \mapsto q'\})}\ Act \qquad \frac{s(Id.\mathbf{loc}) \overset{return}{\nrightarrow}}{s \xrightarrow{\delta,\bot} s \oplus \delta}\ Time$$

$$\frac{t : q \xrightarrow{G/v:=call\ T'(e)/a}_T q' \quad s(Id.\mathbf{templ}) = T \quad s(Id.\mathbf{loc}) = q \qquad s \models G \quad Card\{Id \mid Id.\mathbf{loc} \in dom(s)\} < \mathcal{I}(T') \quad Id' := fresh(s)}{s \xrightarrow{\tau,\bot} s \lhd \{Id.\mathbf{loc} \mapsto \bar{t}\ \} \parallel Init_Id' \parallel fInit^l(Id', T')}\ Call$$

$$\frac{s(Id.\mathbf{loc}) \in \mathcal{D}(s(Id.\mathbf{templ})).F}{s \xrightarrow{\tau/\bot} s/Id}\ Destroy$$

$$\frac{s(Id.\mathbf{loc}) = \bar{t} \quad s(Id'.\mathbf{loc}) = q \quad s(Id'.\mathbf{templ}) = T \qquad q \xrightarrow{G/return\ e/a}_T q' \quad s \models G \quad s(Id'.\mathbf{ParId}) = Id}{s \xrightarrow{\tau,\top} t.a_{Id}(a_{Id'}(s \lhd \{Id.\mathbf{loc} \mapsto t.q', Id'.\mathbf{loc} \mapsto q', t.v \mapsto [\![e]\!]_s^{Id'}\}))}\ Return$$

where $Init_Id' = \{Id'.p_{T'} \mapsto [\![e]\!]_s^{Id}, Id'.\mathbf{ParId} \mapsto Id, Id'.\mathbf{templ} \mapsto T', Id'.\mathbf{loc} \mapsto \mathcal{D}(T').q^0\}$ *is the initialization of parameters and newly created variables of instance* Id' *(rule Call), and the function* $fInit^l(Id', T') =\parallel_{v\in\mathcal{D}(T').V^l} \{Id'.v \mapsto \mathcal{D}(T').Init^l(v)\}$ *is the initialization of the instance original local variables according to the initial valuation* $Init^l$ *of its template* $\mathcal{D}(T')$ *identified by* T'.

The semantics of the CTA instantiation is given through the former definition together with the TTS product. It consists of compiling dynamically CTA instances to TTSs and computing simultaneously the parallel product of these TTSs. In fact, the semantics of a CTA T creates the first instance of T. Such an instance is recognizable by a fresh identifier $Id = fresh(\emptyset)$. The set of local variables of the underlying TTS corresponds to the union of the local variables of all instances of T that can be created according to the maximal number of instances $\mathcal{I}(T)$ i.e., $(\bigcup_{i\in\mathcal{I}(T)}\{Id_i.V^l)$, together with the newly introduced variables $(Id_i.\mathbf{loc}, Id_i.\mathbf{templ}, Id_i.\mathbf{ParId})$. Moreover, TTS states are partial functions where only variables of created instances are valued.

About transitions, rule *Act* states a non-calling transition of an instance Id of template T ($Id.\mathbf{templ} = T$) if the current location of Id corresponds to q. Such a transition is enabled if the current source state s satisfies the guard G, and consists of updating local and global variables according to action a_{Id}, with a jump to location q'. The update action a_{Id} is a rewriting of action a where the local variables of template T, occurring in a are replaced by that of instance Id.

[4] $Id.E = \{Id.e \mid e \in E\}$ consists of prefixing each variables $e \in E$ by the identifier Id of a CTA instance. Such a renaming is used to distinguish between variables of different instances, in particular between instances of the same template where variables have the same original names.

Rule *Time* corresponds to a delay of an instance Id from state s. The notation $q \overset{return}{\nrightarrow}$ states the absence of outgoing transitions labelled with a `return` event, from location q. Implicitly, return events have priority over others. Thus, we do not allow delays from locations having outgoing transitions labelled by a return. Such a restriction is useful to enable instances unlocking their callers once they reach a state having an outgoing return.

After checking that the current location **loc** of an instance Id of template T corresponds to location q, the current state s satisfies the guard G, and the cardinality of the current set of the callee template (T') instances does not cross up the maximal number allowed for this template i.e., $Card\{Id \mid Id.\mathbf{loc} \in dom(s)\} < \mathcal{I}(T')$, rule *Call* creates a new instance Id' of the callee template T'. Such a newly created instance is concurrently run with its calling instance Id of template T, and has the parent (calling) instance identifier $\mathbf{ParId} = Id$. Without executing the update action a, the calling instance Id moves to an intermediate location $\bar{t}$ waiting for a return. The update action a is stored in location $\bar{t}$, and will be applied after assigning the result returned by Id' to variable v. On its creation, the instance Id' initializes its parameter and its new local variables $(\mathbf{loc}, \mathbf{templ}, \mathbf{ParId})$ according to $Init_Id'$, and also initializes its original local variables V^l according to $fInit^l$.

Rule *Destroy* states that an instance Id will be destroyed when it reaches a final location. Such a destruction consists of removing the variables and locations of Id from the system state.

Rule *Return* specifies how an instance Id' of template T performs a return, for its calling instance Id waiting on an intermediate location $\bar{t}$. In fact, after ensuring for whom ($Id'.\mathbf{ParId} = Id$) the return action should be made, the instance Id' yields the result expression e, evaluated to $[\![e]\!]_s^{Id'}$ according to the valuation of state s, to its calling (parent) instance Id. The former joins the target location q', stored in $t.q'$, of its calling transition t after the reception of the returned value $t.v = [\![e]\!]_s^{Id'}$. Through such a transition, from location $\bar{t}$ to $t.q'$, the update action $t.a$ [5] of the transition t, originally performing the call of Id', is applied after the execution of the local action $a_{Id'}$ of the returning transition and the assignment of $[\![e]\!]_s^{Id'}$ to local variable v of Id.

Remark. One may remark that we have unified both static and dynamic instantiations in one semantics. The difference between both instantiation semantics can be clearly distinguished over the following condition $Card\{Id \mid Id.\mathbf{loc} \in dom(s)\} < \mathcal{I}(T')$ of rule *Call*. In fact, in the dynamic instantiation we can create an infinite set of instances because the above condition is always satisfied, i.e., the maximal number ($\mathcal{I}(T') = \infty$) of instances to be created cannot be reached. Whereas in the static instantiation semantics, we are allowed to create a new instance if the number of the current active instances does not cross up the maximal (finite) bound $\mathcal{I}(T')$.

[5] The notation $t.v$ refers to variable v occurring in the left side of the calling transition t label. Similarly, $t.a$ is the update action of transition t.

Theorem 2. (Subprogram call safety) *An instance of a subprogram CTA T is either in its own initial location q^0 or there exists a unique component which is in a waiting location $\bar{t}$ associated to a call to T. Formally, the property P such that $P(s) \equiv \forall Id\ s(Id.\mathbf{loc}) \neq \mathcal{D}(s(Id.\mathbf{templ})).q^0 \Rightarrow \exists! Id'\ \exists! t\ s(Id'.\mathbf{loc}) = \bar{t} \wedge s(Id.\mathbf{ParId}) = Id'$ is an invariant of the system.*

Theorem 3. (Instantiation semantics and translation) *The semantics of a system of CTA and TA, defined by the product of TTS associated to its individual components and that based on the translation of CTA to TA are bisimilar.*

Theorem 4. (Liveness) *For an instance Id, a location q with a call as unique outgoing transition which is locally enabled and such that time elapse is bounded* [6]*, then the call is eventually accepted. Formally, for each calling location q, we have: $(s(Id.\mathbf{loc}) = q) \wedge G \leadsto \exists Id'\ \exists s,\ (Id'.\ \mathbf{ParId} = Id) \wedge (s(Id'.\mathbf{loc}) = \mathcal{D}(s(Id'.\mathbf{templ})).q^0)$, where $\leadsto$ is the UPPAAL **Leads to** operator.*

Theorem 5. *If the NTA translation of a CTA system has always a free instance for each template i.e., $\forall s\ \forall T\ \bigvee_i (s(Id_i.\mathbf{loc}) = \mathcal{D}(T).q^0 \mid Id_i.\mathbf{templ} = T)$, then the TTS associated to the NTA translation and the TTS associated to the dynamic instantiation semantics of the CTA system are bisimilar.*

7 Implementation and Experiments

In order to make our extension profitable, we have designed a PYTHON SCRIPT program converting callable timed automata systems to UPPAAL NTA. In fact, our converter uploads an XML file designed using UPPAAL graphical editor, as an input where the interface of each CTA states a finite number of instances. After performing a deeper analysis of callable automata syntax, in particular template interfaces, *call* and *return* transitions, the converter generates the corresponding UPPAAL NTA format, written in a new XML file that will be then reloaded in the UPPAAL tool, as an ordinary system to be analyzed and checked.

The interface of each callable TA is given by the number of instances, the type of *return*, the name of template and the set of parameters. That is an example of a template signature with 3 instances, a void return type, the template name *Use_Case* and a set of parameters.

```
3;void Use_Case(int ind, int arrival_time, int memory_usb)
```

After replicating template instances in the system declaration, according to the template signatures, the source XML file will be explored template by template and transition by transition. For each callable template occurring in a calling transition, both that transition and the callee template will be translated as stated in Section 4.3.

[6] In UPPAAL, such a property can be enforced by assigning $clock \leq B$ as an invariant to this location.

The converter translates each callable TA, occurring in a calling transition, to a UPPAAL TA by adding an extra synchronizing transition (from q_{init} to q^0) to activate the automaton, another transition (from a final location to q_{init}) to get the instances available for other calls after reaching final locations, a shared variable to hold the name of the current calling template, and splitting each calling transition to a sequence of call and return transitions as shown in Figure 5.

In the case of dynamic interpretation where templates have an infinite number of instances, we infer a finite (sufficient) number simulating the infinite bound of each CTA instantiation as stated in Section 4.3. Then, for each call, we reuse an existing instance instead of creating a new one.

As an application, we have remodeled the `Océ` printer system using callable timed automata, where each job (use-case) is modeled by a callable automaton. We consider 6 templates where only 3 are callable (3 CTA). We have also introduced another template USER to manage the system. The USER triggers dynamically different jobs at different respective dates. We have successfully translated the new model of the `Océ` system to an UPPAAL NTA, and also proceeded on the verification of the property stating that all jobs reach their final locations *DONE*. Such a property is satisfied by both original model [19] and the translation. The `Oce` protocol with CTA is available on http://www.irit.fr/ Abdeldjalil.Boudjadar/EXEMPLES/Oce/oce-model.xml. The corresponding translation is also available on http://www.irit.fr/∼Abdeldjalil.Boudjadar/EXEMPLES/Oce/oce-translation.xml.

8 Conclusion and Perspectives

Throughout this paper, we have introduced and formalized the concept of callable timed automata for the modelling and structuring of real-time and interactive systems. Such a syntactical extension can be interpreted in different semantical ways: static and dynamic. In the dynamic case, we propose to reuse UPPAAL by giving bounds to the numbers of simultaneously active instances of templates. Such a technique can be interesting for the study of population protocols [3] when the population happens to be bounded.

Thanks to the UPPAAL translation, we have validated our proposal through an UPPAAL "plugin".

As a challenging continuation of our work, we envision to consider existing work related to Petri nets as well as to logics that take into account CALL and RETURN like CaRet [2] and Spade [24]. Moreover, we have in mind model checking support for architecture description languages, where subprograms with their own resources are considered [16]. Another point worth studying is related to compositionality. It would be interesting to study how the results of [8] and [11] could be extended to the context of CTA.

References

1. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
2. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)
3. Aspnes, J., Ruppert, E.: An introduction to population protocols. Bulletin of the European Association for Theoretical Computer Science 93, 98–117 (2007)
4. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T.: When are timed automata determinizable? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 43–54. Springer, Heidelberg (2009)
5. Beffara, E.: Functions as proofs as processes. CoRR, abs/1107.4160 (2011)
6. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal 4.0. Department of computer science, Aalborg university (2006)
7. Berendsen, J., Vaandrager, F.: Parallel composition in a paper of Jensen, Larsen and Skou is not associative (2007), Technical note available at http://www.ita.cs.ru.nl/publications/papers/fvaan/BV07.html
8. Berendsen, J., Vaandrager, F.: Compositional Abstraction in Real-Time Model Checking. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 233–249. Springer, Heidelberg (2008)
9. Berendsen, J., Vaandrager, F.: Parallel composition in a paper by de Alfaro e.a. is not associative (2008), Technical note available at http://www.ita.cs.ru.nl/publications/papers/fvaan/BV07.html
10. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool tina-construction of abstract state spaces for petri nets and time petri nets. Intl Journal of Production Research 42 (2004)
11. Bodeveix, J.-P., Boudjadar, A., Filali, M.: An alternative definition for timed automata composition. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 105–119. Springer, Heidelberg (2011)
12. Boudol, G.: Towards a lambda-calculus for concurrent and communicating systems. In: Díaz, J., Orejas, F. (eds.) TAPSOFT 1989. LNCS, vol. 351, pp. 149–161. Springer, Heidelberg (1989)
13. Burns, A., Wellings, A.: Concurrency in Ada, 2nd edn. Cambridge University Press (1998)
14. de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable interfaces. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
15. Engberg, U., Nielsen, M.: A calculus of communicating systems with label passing. Technical report, Computer Science Department, University of Aarhus (1986)
16. Feiler, P.H., Lewis, B., Vestal, S.: The Sae architecture analysis and design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In: RTAS, Workshop, pp. 1–10 (2003)
17. Håkansson, J., Pettersson, P.: Partial order reduction for verification of real-time components. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 211–226. Springer, Heidelberg (2007)
18. Holzmann, G.: Spin model checker, the: primer and reference manual, 1st edn. Addison-Wesley Professional (2003)

19. Igna, G., et al.: Formal modeling and scheduling of datapaths of digital document printers. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 170–187. Springer, Heidelberg (2008)
20. Jensen, H.E., Guldstr, K., Skou, A.: Scaling up Uppaal:Automatic Verification of Real-Time Systems using Compositionality and Abstraction. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 19–30. Springer, Heidelberg (2000)
21. Larsen, K.G., Pettersson, P., Wang, Y.: Uppaal in a nutshell. Journal on Software Tools for Technology Transfert (1997)
22. Milner, R.: Functions as processes. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 167–180. Springer, Heidelberg (1990)
23. Nielson, F.: The typed λ-calculus with first-class processes. In: Odijk, E., Rem, M., Syre, J.-C. (eds.) PARLE 1989. LNCS, vol. 366, pp. 357–373. Springer, Heidelberg (1989)
24. Patin, G., Sighireanu, M., Touili, T.: Spade: Verification of multithreaded dynamic and recursive programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 254–257. Springer, Heidelberg (2007)
25. Thomsen, B.: A calculus of higher order communicating systems. In: Proceedings of the 16th ACM Conference POPL 1989, pp. 143–154. ACM (1989)
26. Toninho, B., Caires, L., Pfenning, F.: Functions as session-typed processes. CoRR (2011), Paper available on http://ctp.di.fct.unl.pt/~lcaires/papers/
27. Trivedi, A., Wojtczak, D.: Recursive timed automata. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 306–324. Springer, Heidelberg (2010)
28. Warn, F.: Red: Model-checker for timed automata with clock-restriction diagram. In: Workshop on Real-time Tools (2001)
29. Yovine, S.: Kronos: A verification tool for real-time systems. Journal of Software Tools for Technology Transfer, 123–133 (1997)

Efficient Operational Semantics for EB^3 for Verification of Temporal Properties

Dimitris Vekris and Catalin Dima

LACL, Université Paris-Est,
61 av. du Général de Gaulle,
94010 Créteil, France
{dimitrios.vekris, dima}@u-pec.fr

Abstract. EB^3 is a specification language for information systems. The core of the EB^3 language consists of process algebraic specifications describing the behaviour of the entity types in a system, and attribute function definitions describing the entity attribute types. The verification of EB^3 specifications against temporal properties is of great interest to users of EB^3. We give here an operational semantics for EB^3 programs in which attribute functions are computed during program evolution and their values are stored into program memory. By assuming that all entities have finite domains, this gives a finitary operational semantics. We then demonstrate how this new semantics facilitates the translation of EB^3 specifications to LOTOS NT (LNT for short) for verification of temporal properties with the use of the CADP toolbox.

Keywords: Information Systems, EB^3, Process Algebras, Operational Semantics, Bisimulation, Verification, Model Checking.

1 Introduction

The EB^3 [10] method is an event-based paradigm for information systems (ISs) [17]. A typical EB^3 specification defines entities, associations, and their respective attributes. The process algebraic nature of EB^3 permits the explicit definition of intra-entity constraints. Yet its specificity against common state-space specifications, such as the *B method* [1] and *Z*, lies in the use of attribute functions, a special kind of recursive functions on the system trace, which combined with guards, facilitate the definition of complex inter-entity constraints involving the history of events. The use of attribute functions is claimed to simplify system understanding, enhance code modularity and streamline maintenance.

In this paper, we present part of our work regarding the verification of EB^3, i.e. the detection of errors inherent in EB^3 specifications. Specification errors in EB^3 can be detected with the aid of static properties also known as invariants or dynamic properties known as temporal properties. From a state-based point of view, an invariant describes a property on state variables that must be preserved by each transition or event. A temporal property relates several events. Tools such as *Atelier B* [7] provide methodologies on how to define and prove

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 133–149, 2013.
DOI: 10.1007/978-3-642-40213-5_9,
© IFIP International Federation for Information Processing 2013

invariants. In [12], an automatic translation of EB^3's attribute functions into B is attempted. Although the *B Method* [1] is suitable for specifying static properties, temporal properties are very difficult to express and verify in B. Hence, in our attempt to verify temporal properties of EB^3 specifications we move our attention to model-checking techniques.

The verification of EB^3 specifications against temporal properties with the use of model checking has been the subject of some work in the recent years. [9] compares six model checkers for the verification of IS case studies. The specifications used in [9] derive from specific industrial case studies, but the prospect of a uniform translation from EB^3 program specifications is not studied. [6] casts an IS specification into LOTOS NT (LNT for short) [5] that serves as an input language to the verification suite CADP [11]. In short, the majority of these works treat specific case studies drawn from the information systems domain leading to ad-hoc verification translations, but nonetheless lacking in generalization capability.

But the main problem in verifying EB^3 specifications against temporal-logic properties relies in the difficulty to handle the recursive definition of attribute functions if one relies on the classical, trace-based semantics. This type of semantics necessitates an unbounded memory model, and therefore only bounded model-checking can be achieved, in the absence of good abstractions that allow constructing finite-state models. This restriction is present in the original approach [10] and the subsequent model-checking attempt [9] even if all the entities utilized in the specification are finite.

We propose a formal semantics for EB^3 that treats attribute functions as state variables (we call these variables *attribute variables*). This semantics will serve as the basis for applying a simulation strategy of state variables in LNT. Intuitively, coding attribute functions as part of the system state is beneficial from a model-checking point of view as the new formalisation dispenses with the system trace. Our main contribution is an operational semantics in which attribute functions are computed during program evolution and stored into program memory. We show that this operational semantics is bisimilar with the original, trace-based operational semantics.

Furthermore, we explore the implications of this result to the translation of EB^3 specifications into LNT. LNT is a process algebra specification that derived from LOTOS [4]. As a process algebra, it shares many common features with EB^3 and it is one of the input languages of CADP, a toolbox with state-of-the-art verification features. CADP permits the verification of system specifications against action-based temporal properties.

Translating EB^3 specifications to LNT is not evident. The fundamental difficulties for designing a compiler from EB^3 into LNT are summarized in [6]. In particular, LNT does not feature global variables. Accesses to local variables is restricted in parallel processes of the form "**par** $proc_1$ || $proc_2$ **end par**", so that every variable written in $proc_1$ cannot be accessed in $proc_2$. Although, EB^3 programmers cannot define global variables explicitly, EB^3 permits the use of a single state variable, the system trace, in predicates of guard statements.

Attribute functions can express the evolution of entity attributes in time, option which introduces an indirect notion of state to the language. As a result, EB^3 expressions of the form "$C(T) \Rightarrow E$" can be written, where $C(T)$ is a predicate that refers to the system trace (the history of events) and E is a valid EB^3 expression.

We then present how EB^3 specifications can be translated to LNT for verification with CADP through an intuitive example and give some conclusions and lines for future work. The automatic translation of EB^3 specifications into LNT is studied in the companion paper [18]. We note that the translation of our example into LNT is produced using the tool presented in [18].

2 EB^3

The EB^3 method has been specially designed to specify the functional behaviour of ISs. A standard EB^3 specification comprises (1) a class diagram representing entity types and associations for the IS being specified, (2) a process algebra specification, denoted by *main*, describing the IS, i.e. the valid traces of execution describing its behaviour, (3) a set of attribute function definitions, which are recursive functions on the system trace, and (4) input/output rules, to specify outputs for input traces, or SQL used to specify queries on the business model. We limit the presentation to the process algebra and the set of attribute functions used in the IS.

We then give three operational semantics for EB^3. The first, named Trace Semantics (Sem_T), is the standard semantics defined in [10]. The second, called Trace/Memory Semantics ($Sem_{T/M}$), is the alternative semantics, where attribute functions are computed during program evolution and their values are stored into program memory. By removing the trace from each state in $Sem_{T/M}$, we obtain the third semantics for EB^3 specifications, which we name Memory Semantics, Sem_M. The relevance of the $Sem_{T/M}$ semantics stems from the fact that it is pivotal in proving the bisimulation between Sem_T and Sem_M.

Case Study. We start by providing a simple case study which serves for introducing both the syntax and the semantics of EB^3. In Fig. 1, we give the functional requirements of a library management system and the corresponding EB^3 specification.The library system contains two entity types: *books* and *members*. The process *main* is the parallel interleaving between m instances of process *book* and p instances of processes describing operations on *members*. To avoid confusion, action names begin with uppercase letters, while process and attribute function names begin with lowercase letters.

The member *mId* registers to the library in order to start borrowing books, i.e. the action *Register*(*mId*). By the action *Unregister*(*mId*), (s)he relinquishes membership from the library. The book *bId* is acquired by the library so as to become available for lending, i.e. *Acquire*(*bId*). The inverse operation is expressed by the action *Discard*(*bId*). The member *mId* borrows the book *bId*, i.e. *Lend*(*bId*, *mId*) and returns it to the library after use, i.e. *Return*(*bId*). The process *book*(*bId*) denotes the lifecycle of the *book* entity *bId* from the

1. A book can be acquired by the library. It can be discarded, but only if it has not been lent.
2. An individual must join the library in order to borrow a book.
3. A member can relinquish library membership only when all his loans have been returned.
4. A member cannot borrow more than the loan limit defined at the system level for all users.

$$
\begin{aligned}
BID &= \{b_1, \ldots, b_m\},\ MID = \{m_1, \ldots, m_p\} \\
main &= (\ |||\ bId : \text{BID} : book(bId)\ )\ |||\ (\ |||\ mId : \text{MID} : member(mId)^*\ ) \\
book(bId : \text{BID}) &= Acquire(bId).\ borrower(T,\ bId) = \bot \Rightarrow Discard(bId) \\
member(mId : \text{MID}) &= Register(mId).\ (\ |||\ bId : \text{BID} : loan(mId, bId)^*\ ).\ Unregister(mId) \\
loan(mId : \text{MID},\ bId : \text{BID}) &= borrower(T,\ bId) = \bot \wedge nbLoans(T, mId) < \text{NbLoans} \\
&\quad \Rightarrow Lend(bId,\ mId).\ Return(bId)
\end{aligned}
$$

```
nbLoans(T: tr, mId: MID): Nat⊥ =
  match T with
    [ ] → ⊥
  | T'. Lend(bId, mId) → nbLoans(T', mId) + 1
  | T'. Register(mId) → 0
  | T'. Unregister(mId) → ⊥
  | T'. Return(bId) ∧ mId = borrower(T, bId)
      → nbLoans(T', mId) − 1
  | T'._ → nbLoans(T', mId)
  end match
```

```
borrower(T: tr, bId: BID): MID⊥ =
  match T with
    [ ] → ⊥
  | T'. Lend(bId, mId) → mId
  | T'. Return(bId) → ⊥
  | T'._ → borrower(T', bId)
  end match
```

Fig. 1. EB^3 Specification and Attribute Function Definitions

moment of its acquisition until its eventual discard from the library. The process $member(mId)$ denotes the lifecycle of the *member* entity mId from the point of its registration up until its membership drop. In the body of $member(mId)$, the process expression "$|||bId : BID : loan(mId, bId)^*$" denotes the interleaving of m instances of the process expression $loan(mId, bId)^*$ that, according to the standard semantics of the *Kleene Closure* operator (*), denotes the execution of $loan(mId, bId)$, $bId = \{b_1, \ldots, b_m\}$ an arbitrary, but bounded number of times. The attribute function $borrower(T, bId)$, where T is the current trace, returns the current borrower of book bId or $\bot$ (meaning *undefined*) if the book is not lent, by looking for actions of the form $Lend(bId, mId)$ or $Return(bId)$ in the trace. In process $book(bId)$, the action $Discard(bId)$ is thus guarded by $borrower(T, bId) = \bot$ to guarantee that the book bId cannot be discarded if it is currently lent.

The use of attribute functions is not adherent to standard process algebra practices as it may naively trigger the complete traversal and inspection of the system trace. Alternatively, one may come up with simpler specifications based solely on process algebra operations (without attribute functions) when the functional requirements imply loose interdependence between entities and associations. For instance, if all books are acquired by the library before any other action occurs and are eventually discarded (given that there are no more demands), *main*'s code can be modified in the following manner:

$$
\begin{aligned}
main = &(\ |||\ bId : \text{BID} : Acquire(bId)\ ).\ (\ |||\ mId : \text{MID} : member(mId)^*\ ). \\
&(\ |||\ bId : \text{BID} : Discard(bId)\ )
\end{aligned}
$$

Note that the functional requirements are not contradicted, though the system's behaviour changes dramatically. Programming naturally in a purely process-algebraic style without attribute functions in EB^3 may not always be

$$main \qquad (A)$$

$$\xrightarrow{Acq(b_2).Acq(b_1)}$$

$$borrower(T,\ b_1) = \bot \rightarrow Discard(b_1) \mid\mid\mid$$
$$borrower(T,\ b_2) = \bot \rightarrow Discard(b_2) \mid\mid\mid$$
$$(\ \mid\mid\mid mId : \text{MID} : member(mId)^*\ ) \qquad (B)$$

$$\xrightarrow{Reg(m_2).Reg(m_1)}$$

$$borrower(T,\ b_1) = \bot \rightarrow Discard(b_1) \mid\mid\mid$$
$$borrower(T,\ b_2) = \bot \rightarrow Discard(b_2) \mid\mid\mid$$
$$(\ \mid\mid\mid bId : \text{BID} : loan(m_1, bId)^*\ ).\ Unregister(m_1).\ member(m_1)^* \mid\mid\mid$$
$$(\ \mid\mid\mid bId : \text{BID} : loan(m_2, bId)^*\ ).\ Unregister(m_1).\ member(m_2)^* \qquad (C)$$

$$\xrightarrow{Lend(b_1,\ m_1)}$$

$$borrower(T,\ b_1) = \bot \rightarrow Discard(b_1) \mid\mid\mid$$
$$borrower(T,\ b_2) = \bot \rightarrow Discard(b_2) \mid\mid\mid$$
$$(Return(b_1).\ loan(m_1, b_1)^* \mid\mid\mid loan(m_1, b_2)^*).Unregister(m_1).\ member(m_1)^* \mid\mid\mid$$
$$(\ \mid\mid\mid bId : \text{BID} : loan(m_2, bId)^*\ ).\ Unregister(m_1).\ member(m_2)^* \qquad (D)$$

Fig. 2. Sample execution

obvious. In some cases, ordering constraints involving several entities are quite difficult to express without guards and lead to less readable specifications than equivalent guard-oriented solutions in EB^3 style. For instance in the body of $loan(mId : MID, bId : BID)$, writing the specification without the use of the guard:

$$borrower(T,\ bId) = \bot \wedge nbLoans(T, mId) < \text{NbLoans}$$

that illustrates the conditions under which a *Lend* can occur (notably when the book is available and *nbLoans* is less than the fixed bound *NbLoans*), is not trivial.

Execution. As a means to provide the operational intuition behind the three semantics introduced later in this section, we show how the EB^3 specification above is transformed through a four-step trace, assuming that the library may contain at most two books and at most two members, that is, $BID = \{b_1, b_2\}$ and $MID = \{m_1, m_2\}$.

First, we associate with the attribute function *borrower* two "memory cells", $bor[b_1]$ and $bor[b_2]$, meant to encode the value computed by the function for each book ID after each trace T. Similarly, we associate two memory cells $nbL[m_1], nbL[m_2]$ to the attribute function *nbLoans*. We also set $NbLoans = 2$ for the constant used in the definition of the process term *loan*.

	T	$M = (bor[b_1], bor[b_2], nbL[m_1], nbL[m_2])$
A	$[\,]$	$(\bot, \bot, \bot, \bot)$
B	$Acq(b_2).Acq(b_1)$	$(\bot, \bot, \bot, \bot)$
C	$T_B.Reg(m_2).Reg(m_1)$	$(\bot, \bot, 0, 0)$
D	$T_C.Lend(b_1, m_1)$	$(m_1, \bot, 1, 0)$

Fig. 3. States for the sample execution

Figure 2 shows how the process term *main* evolves by executing the valid trace $T_D = Acq(b_2).Acq(b_1).Reg(m_2).Reg(m_1).Lend(b_1, m_1)$, in which *Acq* stands for

Acquire and *Reg* for *Register*, respectively. During this evolution, the two attribute functions are computed according to their specifications in Fig. 2, inductively on the length of the trace. Hence, initially and after the execution of actions $Acq(b_2).Acq(b_1)$, the two attribute functions are undefined for both their arguments, while after the execution of the sequence of actions $Reg(m_2).Reg(m_1)$ we have "$nbLoans(T[C], m_1) = nbLoans(T[C], m_2) = 0$" and $borrower(T, .)$ remains undefined. These values are employed in order to check "$borrower(T, b_1) = \perp \wedge nbLoans(T, m_1) < 2$", which leads to the possibility for member m_1 to lend book b_1, and therefore to transform the process term at (C) in the process term at (D).

On the other hand, the table in Fig. 3 indicates the memory status after each (pair of) actions in the given trace. Initially, all memory cells carry the undefined value. After the trace $T[C]$, the value of memory cell $nbL_C[m_1]$ equals $nbLoans(T[C], m_1)$, and, similarly, "$nbL_C[m_2] = nbLoans(T[C], m_2)$". Note that the constraint checked at step $C \rightarrow D$ gives the same value regardless of the utilization of the value computed recursively for the attribute functions *borrower* and *nbLoans*, or by using the corresponding memory cells. Furthermore, the execution of action $Lend(b_1, m_1)$ triggers the update of the memory cell $bor[b_1]$ to m_1 and the incrementation of $nbL[m_1]$ to 1. This is modeled by the application of a function *next*, which defines the evolution of the system memory, and which is defined as follows: "$bor_D[b_1] = next(bor_C[b_1])$ [1]$= m1$ [2]","$bor_D[b_2] = bor_C[b_2] = \perp$", "$nbL_D[m_1] = nbL_C[m_1] + 1 = 1$" and also "$nbL_D[m_2] = 0$".

$\mathbf{EB^3}$ Syntax and $\mathbf{Sem_T}$. We proceed with the formal definition of EB^3. We define a set of attribute function names $AtFct = \{f_1, \ldots, f_n\}$ and a set of process function names $PFct = \{P_1, \ldots, P_m\}$. Let $\rho \in Act$ stand for an action of either form $\alpha(p_1 : T_1, \ldots, p_n : T_n)$, where $\alpha \in lab$ [3] is the *label* of the action and p_i, $i \in 1..n$ are elements of type T_i, or λ, which stands for the internal action. To simplify the presentation, we assume that all attribute functions f_i have the same formal parameters $\overline{x}$. An EB^3 specification is a set of attribute function definitions AtF and a set of process definitions $ListPE$.

Sem_T [10] is given in Fig. 4 as a set of rules named $R_T{-}1$ to $R_T{-}11$. Each state is represented as a tuple (E, T), where E stands for an EB^3 expression and T for the current trace. An action ρ is the simplest EB^3 process, whose semantics are given by rules $R_T - 1, 1'$. Note that λ is not visible in the EB^3 execution trace, i.e., it does not impact the definition of attribute functions. The symbol $\surd$ denotes successful execution. EB^3 processes can be combined with classical process algebra operators such as the *sequence* ($R_T{-}2, 3$), the *choice* ($R_T{-}4$) and the *Kleene* Closure ($R_T{-}5, 6$) operators. Rules ($R_T{-}7, 8, 9$) refer to the *parallel* composition $E_1|[\Delta]|E_2$ of E_1, E_2 with synchronization on $\Delta \subseteq lab$. The condition $in(\rho, \Delta)$ is true, iff the label of ρ belongs to Δ. The symmetric rules

[1] here notation $next(x_C)$ denotes the modification on x's value in state (C) after executing transition $C \rightarrow D$

[2] see *borrower*'s script for "$T = T'.Lend(bId, mId)$" in Fig. 1

[3] we assume $lab = \{\alpha_1, \ldots, \alpha_q\}$

$$EB^3 ::= AttrF \ ; \ ListPE$$
$$ListPE ::= P_l(\overline{x_l}) = E \ \Big| \ P_l(\overline{x_l}) = E \ ; \ ListPE, \ l \in 1..m$$
$$AtF ::= AtFDef \ \Big| \ AtFDef \ ; \ AtF$$
$$AtFDef ::= f_i(T, \overline{x}) = \begin{cases} exp_i^0 & \text{if } T = [\,] \\ \bigvee_{j=1}^{q} \bigvee_{k=1}^{m_j} hd(T) = \alpha_j(\overline{x_j}) \wedge cond_i^{j,k} \Rightarrow exp_i^{j,k} & \text{otherwise} \end{cases}, \ i \in 1..n$$
$$E ::= \surd \ \Big| \ \lambda \ \Big| \ \alpha(\overline{v}) \ \Big| \ E.E \ \Big| \ E|E \ \Big| \ E^* \ \Big| \ E|[\Delta]|E \ \Big| \ |x:V:E \ \Big| \ |[\Delta]|x:V:E \ \Big| \ GE \Rightarrow E \ \Big| \ P(\overline{t})$$

$$R_T-1: \frac{}{(\rho, T) \xrightarrow{\rho} (\surd, T\cdot\rho)} \rho \neq \lambda \qquad R_T-1': \frac{}{(\lambda, T) \xrightarrow{\lambda} (\surd, T)}$$

$$R_T-2: \frac{(E_1, T) \xrightarrow{\rho} (E_1', T')}{(E_1.E_2, T) \xrightarrow{\rho} (E_1'.E_2, T')} \qquad R_T-3: \frac{(E, T) \xrightarrow{\rho} (E', T')}{(\surd.E, T) \xrightarrow{\rho} (E', T')}$$

$$R_T-4: \frac{(E_1, T) \xrightarrow{\rho} (E_1', T')}{(E_1|E_2, T) \xrightarrow{\rho} (E_1', T')} \qquad R_T-5: \frac{}{(E^*, T) \xrightarrow{\lambda} (\surd, T)}$$

$$R_T-6: \frac{(E, T) \xrightarrow{\rho} (E', T')}{(E^*, T) \xrightarrow{\rho} (E'.E^*, T')} \qquad R_T-7: \frac{}{(\surd|[\Delta]|\surd, T) \xrightarrow{\lambda} \surd, T)}$$

$$R_T-8: \frac{(E_1, T) \xrightarrow{\rho} (E_1', T'), \ (E_2, T) \xrightarrow{\rho} (E_2', T')}{(E_1|[\Delta]|E_2, T) \xrightarrow{\rho} (E_1'|[\Delta]|E_2', T')} in(\rho, \Delta)$$

$$R_T-9: \frac{(E_1, T) \xrightarrow{\rho} (E_1', T')}{(E_1|[\Delta]|E_2, T) \xrightarrow{\rho} (E_1'|[\Delta]|E_2, T')} \neg in(\rho, \Delta)$$

$$R_T-10: \frac{(E, T) \xrightarrow{\rho} (E', T')}{(C\,(T) \Rightarrow E, T) \xrightarrow{\rho} (E', T')} \ \|C\,(T)\|$$

$$R_T-11: \frac{(E[\overline{x} := \overline{t}], T) \xrightarrow{\rho} (E', T')}{(P(\overline{t}), T) \xrightarrow{\rho} (E', T')} P(\overline{x}) = E \in ListPE$$

Fig. 4. EB^3 Syntax and Sem_T

for *choice* and *parallel* composition have been omitted. Expression $E_1|||E_2$ is equivalent to $E_1|[\emptyset]|E_2$ and $E_1||E_2$ to $E_1|[lab]|E_2$.

In R_T-10, the *guarded expression* process "$C\,(T) \Rightarrow E$" can execute E if the predicate $C\,(T)$ holds. The syntax of $C(T)$ is given below:

$$C\,(T) ::= true \mid false \mid op\,(C\,(T), \ldots, C\,(T)) \mid f_i\,(T, \ldots),\ i \in 1..n,\ op \in \{\wedge, \vee\}$$

This syntax is simplified in the sense that certain expressions cannot be supported in practice, e.g. "$nbLoans(T, mId) < NbLoans$" in Fig. 1. To palliate this, we need to add an attribute function name *nbLoans_lt_NbLoans* to *AtFct* and a corresponding attrribute function definition that implements this inequality. Finally, "$nbLoans(T, mId) < NbLoans$" has to be replaced by *nbLoans_lt_NbLoans* in the EB^3 specification. Note also that this syntax makes strictly use of those "$f_i\,(T, \ldots)$, $i \in 1..n$" with Boolean return-type. Thus, the interpretation function of *guarded expressions* $\|\,.\,\|$ is the standard Boolean interpretation.

Quantification is permitted for *choice* and *parallel* composition. If V is a set of attributes $\{t_1, \ldots, t_n\}$, $|x : V : E$ and $|[\Delta]|x : V : E$ stand respectively for

$$M_i^0(\overline{x}) = \|exp_i^0(\overline{x})\|$$
$$next(M_i)(\rho_j)(\overline{x}) = \|exp_i^{j,k}(\overline{x})[f_l \leftarrow \text{if } l < i \text{ then } next(M_l)(\rho_j) \text{ else } M_l]\|,$$
$$\text{if } \|cond_i^{j,k}(\overline{x})[f_l \leftarrow \text{if } l < i \text{ then } next(M_l) \text{ else } M_l]\|, \ i \in 1..n\ 1, \ k \in 1..m_j$$

$$T_{T/M}-1: \frac{\rho \neq \lambda}{(\rho, T, M) \xrightarrow{\rho} (\surd, T\cdot\rho, next(M)(\rho))} \quad T_{T/M}-1': \frac{}{(\lambda, T, M) \xrightarrow{\lambda} (\surd, T, M)}$$

$$T_{T/M}-2: \frac{(E_1, T, M) \xrightarrow{\rho} (E_1', T', M')}{(E_1.E_2, T, M) \xrightarrow{\rho} (E_1'.E_2, T', M')} \quad T_{T/M}-3: \frac{(E, T, M) \xrightarrow{\rho} (E', T', M')}{(\surd.E, T, M) \xrightarrow{\rho} (E', T', M')}$$

$$T_{T/M}-4: \frac{(E_1, T, M) \xrightarrow{\rho} (E_1', T', M')}{(E_1|E_2, T, M) \xrightarrow{\rho} (E_1', T', M')} \quad T_{T/M}-5: \frac{}{(E^*, T, M) \xrightarrow{\lambda} (\surd, T, M)}$$

$$T_{T/M}-6: \frac{(E, T, M) \xrightarrow{\rho} (E', T', M')}{(E^*, T, M) \xrightarrow{\rho} (E'.E^*, T', M')} \quad T_{T/M}-7: \frac{}{(\surd|[\Delta]|\surd, T, M) \xrightarrow{\lambda} \surd, T, M)}$$

$$T_{T/M}-8: \frac{(E_1, T, M) \xrightarrow{\rho} (E_1', T', M'), \ (E_2, T, M) \xrightarrow{\rho} (E_2', T', M')}{(E_1|[\Delta]|E_2, T, M) \xrightarrow{\rho} (E_1'|[\Delta]|E_2', T', M')} in(\rho, \Delta)$$

$$T_{T/M}-9: \frac{(E_1, T, M) \xrightarrow{\rho} (E_1', T', M')}{(E_1|[\Delta]|E_2, T, M) \xrightarrow{\rho} (E_1'|[\Delta]|E_2, T', M')} \neg in(\rho, \Delta)$$

$$T_{T/M}-10: \frac{(E, T, M) \xrightarrow{\rho} (E', T', M')}{(C\,(T) \Rightarrow E, T, M) \xrightarrow{\rho} (E', T', M')} \ \|C[f_i \leftarrow M_i]\|$$

$$T_{T/M}-11: \frac{(E[\overline{x} := \overline{t}], T, M) \xrightarrow{\rho} (E', T', M')}{(P(\overline{t}), T, M) \xrightarrow{\rho} (E', T', M')} P(\overline{x}) = E \in ListPE$$

Fig. 5. $Sem_{T/M}$

$E[x := t_1]| \dots |E[x := t_n]$ and $E[x := t_1]|[\Delta]| \dots |[\Delta]|E[x := t_n]$, where $E[x := t]$ denotes the replacement of all occurrences of x by t. For instance, $||x : \{1, 2, 3\} : a(x)$ stands for $a(1)||a(2)||a(3)$. By convention, $|x : \emptyset : E = |[\Delta]|x : \emptyset : E = \surd$.

Attribute functions are defined in *AtFDef* [4] in Fig. 4, where $exp_i^{j,k}$ are expressions, $cond_i^{j,k}$ are boolean expressions, $hd(T)$ denotes the last element of the trace, and $tl(T)$ denotes the trace without its last element. Expressions can be constructed from objects and operations of user-defined domains, such as integers, booleans and more complex domains that we do not give formally. We also assume that for each $1 \leq i \leq n$, all calls to an attribute function f_l occurring in $exp_i^{j,k}$ or $cond_i^{j,k}$ are parameterized by T if $l \leq i$ or by $tl(T)$ if $l > i$. Such an ordering can be constructed if the EB^3 specification does not contain circular dependencies between function calls, which would lead to infinite attribute function evaluation. This restriction on *AtFct* is satisfied in our case study as both *nbLoans* and *borrower* contain calls to *nbLoans* and *borrower* parameterized on $tl(T)$. Also, *nbLoans* makes call to *borrower* parameterized on T. Hence, $f_1 = borrower$ and $f_2 = nbLoans$.

$\mathbf{Sem_{T/M}}$. $Sem_{T/M}$ is given in Fig. 5 as a set of rules named $T_{T/M}-1$ upto $T_{T/M}-11$. Each state is represented as a tuple (E, T, M). $M_i(\overline{x})$ is the variable

[4] This notation is different from the standard pattern-matching notation for attribute functions [10], but more compact

$$S_M-1: \frac{\rho \neq \lambda}{(\rho, M) \xrightarrow{\rho} (\surd, next(M)(\rho))} \quad S_M-1': \frac{}{(\lambda, M) \xrightarrow{\lambda} (\surd, M)}$$

$$S_M-2: \frac{(E_1, M) \xrightarrow{\rho} (E_1', M')}{(E_1.E_2, M) \xrightarrow{\rho} (E_1'.E_2, M')} \quad S_M-3: \frac{(E, M) \xrightarrow{\rho} (E', M')}{(\surd.E, M) \xrightarrow{\rho} (E', M')}$$

$$S_M-4: \frac{(E_1, M) \xrightarrow{\rho} (E_1', M')}{(E_1|E_2, M) \xrightarrow{\rho} (E_1', M')} \quad S_M-5: \frac{}{(E^*, M) \xrightarrow{\lambda} (\surd, M)}$$

$$S_M-6: \frac{(E, M) \xrightarrow{\rho} (E', M')}{(E^*, M) \xrightarrow{\rho} (E'.E^*, M')} \quad S_M-7: \frac{}{(\surd|[\Delta]|\surd, M) \xrightarrow{\lambda} \surd, M)}$$

$$S_M-8: \frac{(E_1, M) \xrightarrow{\rho} (E_1', M'),\ (E_2, M) \xrightarrow{\rho} (E_2', M')}{(E_1|[\Delta]|E_2, M) \xrightarrow{\rho} (E_1'|[\Delta]|E_2', M')} in(\rho, \Delta)$$

$$S_M-9: \frac{(E_1, M) \xrightarrow{\rho} (E_1', M')}{(E_1|[\Delta]|E_2, M) \xrightarrow{\rho} (E_1'|[\Delta]|E_2, M')} \neg in(\rho, \Delta)$$

$$S_M-10: \frac{(E, M) \xrightarrow{\rho} (E', M')}{(C \Rightarrow E, M) \xrightarrow{\rho} (E', M')} \ \|C[f_i \leftarrow M_i]\|$$

$$S_M-11: \frac{(E[\overline{x} := \overline{t}], M) \xrightarrow{\rho} (E', M')}{(P(\overline{t}), M) \xrightarrow{\rho} (E', M')} P(\overline{x}) = E \in ListPE$$

Fig. 6. Sem_M

that keeps the current valuation for attribute function f_i with parameter vector $\overline{x}$. M_i refers to attribute function f_i. Given that the EB^3 specification is valid, there is at least one $cond_i^{j,k}$ that is evaluated true on every run. The action ρ_j to occur "chooses" the corresponding $cond_i^{j,k}$ non-deterministically (in the sense that there may be many k that make $cond_i^{j,k}$ evaluate to true). Function *next* updates M_i by making use of M_l for $l \geq i$ and the freshly computed $next(M_l)(\rho_j)$ for $l < i$. The classic intepretations for Peano Arithmetic, Set Theory and Boolean Logic suffice to evaluate them. In $(T_{T/M}-10)$, $C[f_i \leftarrow M_i]$ denotes replacing all calls to f_i in C by M_i. The notation $\|.\|$ in $\| C[f_i \leftarrow M_i] \|$ corresponds to the standard interpretation of Boolean operators.

$\mathbf{Sem_M}$. Sem_M is given in Fig. 6 as a set of rules named S_M-1 to S_M-11. Sem_M derives from $Sem_{T/M}$ by simple elimination of T from each tuple (E, T, M) in rules $T_{T/M}-1$ upto $T_{T/M}-11$. It gives a finite state system. Intuitively, this means that the information on the history of executions is kept in M, thus rendering the presence of trace T redundant.

3 Bisimulation Equivalence of Sem_T, $Sem_{T/M}$ and Sem_M

We present the proof of the bisimulation equivalence for the three semantics: Sem_T, $Sem_{T/M}$ and Sem_M.

LTSs. We consider finite labeled transition systems (LTSs) as interpretation models, which are particularly suitable for action-based description formalisms such as EB^3. Formally, an LTS is a triple $(S, \{\xrightarrow{a}\}_{a \in Act}, I)$, where: (1) S is a set of states, (2) $\xrightarrow{a} \subseteq S \times S$, for all $a \in Act$, (3) $I \subseteq$ S is a set of initial states.

Bisimulation. Bisimulation is a fundamental notion in the framework of concurrent processes and transition systems. A system is bisimilar to another system if the former can mimic the behaviour of the latter and vice-versa. In this sense, the associated systems are considered indistinguishable. Given two LTSs $TS_i = (S_i, \{\xrightarrow{a}_i\}_{a \in Act}, I_i)$, where $i = 1, 2$ and a relation $R \subseteq S_1 \times S_2$, R is said to be a bisimulation and TS_i are said to be equivalent w.r.t. bisimulation iff

1. $\forall\, s_1 \in I_1\ \exists s_2 \in I_2$ such that $(s_1, s_2) \in R$.
2. $\forall\, s_2 \in I_2\ \exists s_1 \in I_1$ such that $(s_1, s_2) \in R$.
3. $\forall (s_1, s_2) \in R$:
   (a) if $s_1 \xrightarrow{a}_1 s_1'$ then $\exists s_2' \in S_2$ such that $s_2 \xrightarrow{a}_2 s_2'$ and $(s_1', s_2') \in R$;
   (b) if $s_2 \xrightarrow{a}_2 s_2'$ then $\exists s_1' \in S_1$ such that $s_1 \xrightarrow{a}_1 s_1'$ and $(s_1', s_2') \in R$.

LTS Construction. For a given EB^3 process E, we associate three LTSs w.r.t. Sem_T, $Sem_{T/M}$ and Sem_M respectively. These correspond to the LTSs generated inductively by the rules given in Fig. 4–6. The whole process mimicks the construction of a transition system associated with a *transition system specification*, as in [16]. For the rest, we denote TS_T, $TS_{T/M}$ and TS_M for TS_E w.r.t. Sem_T, $Sem_{T/M}$ and Sem_M respectively.

Theorem 1. *TS_T and $TS_{T/M}$ are equivalent w.r.t. bisimulation.*

Proof. Let $\rightarrow_1$ be the transition relation for TS_T and $\rightarrow_2$ be the transition relation for $TS_{T/M}$. The relation, which will give the bisimulation between TS_T and $TS_{T/M}$, is: $R = \{\langle (E, T, M), (E, T)\rangle \mid (E, T, M) \in S_{T/M} \wedge (E, T) \in S_T\}$. Note first that $\langle (E^0, [\,], M^0), (E^0, [\,])\rangle \in R$. We show that for any $\langle (E, T, M), (E, T)\rangle \in R$ and $(E, T, M) \xrightarrow{\rho}_1 (E', T', M') \in \delta_{T/M}$, we obtain $(E, T) \xrightarrow{\rho}_2 (E', T') \in \delta_T$ and vice-versa. We proceed with structural induction on E and present the proof for some cases.

For $(T_{T/M} - 1)$, suppose $(\rho, T, M) \xrightarrow{\rho}_1 (\surd, T \cdot \rho, next(M)(\rho)) \in \delta_{T/M}$. The rule $(R_T - 1)$ allows us to conclude that also $(\rho, T) \xrightarrow{\rho}_2 (\surd, T \cdot \rho) \in \delta_T$. Conversely, suppose $(\rho, T) \xrightarrow{\rho}_2 (\surd, T \cdot \rho) \in \delta_T$. Note that each state $(E, T, M) \in S_{T/M}$ is of the form:

$$
\begin{gathered}
(E, T, next'(T, M^0)), \text{where} \\
next'\,(T, M) = \textbf{match}\ T\ \textbf{with}\ [\,] \rightarrow\ M \mid T' \cdot \rho \rightarrow next'\,(T', next\,(M, \rho))
\end{gathered}
$$

Thus, there exists $(\rho, T, next'\,(T, M^0)) \xrightarrow{\rho}_1 (\surd, T \cdot \rho, next'\,(T \cdot \rho, M^0)) \in \delta_{T/M}$, which establishes rule $(T_{T/M} - 1)$ by replacing $next'(T, M^0)$ with M as well as $next'(T \cdot \rho, M^0)$ with $next\,(M)\,(\rho)$.

For $(T_{T/M} - 2)$, suppose $(E_1 . E_2, T, M) \xrightarrow{\rho}_1 (E_1' . E_2, T', M') \in \delta_{T/M}$, which relies on the existence of a transition $(E_1, T, M) \xrightarrow{\rho}_1 (E_1', T', M') \in \delta_{T/M}$. By the induction hypothesis, $(E_1, T) \xrightarrow{\rho}_2 (E_1', T') \in Sem_T$ and by $(R_T - 2)$, we get $(E_1 . E_2, T) \xrightarrow{\rho}_2 (E_1' . E_2, T') \in \delta_T$. Vice-versa, by virtue of $(R_T - 2)$ a transition $(E_1 . E_2, T) \xrightarrow{\rho}_2 (E_1' . E_2, T') \in \delta_T$ necessitates $(E_1, T) \xrightarrow{\rho}_2 (E_1', T') \in \delta_T$. Using

the induction hypothesis, $(E_1, T, M) \xrightarrow{\rho}_1 (E'_1, T', M')$. Finally, by $(T_{T/M} - 2)$ we obtain $(E_1.E_2, T, M) \xrightarrow{\rho}_1 (E'_1.E_2, T', M')$.

For $(T_{T/M} - 10)$, we must prove that $\| C(T) \| = \| C[f_i \leftarrow M_i] \|$. Making use of the syntactic definition of $C(T)$ and the interpretation of $\| . \|$, it suffices to prove that $f_i(T, \overline{x}) = M_i(\overline{x}), i \in 1..n$ for any parameter vector $\overline{x}$ and trace T. We prove this by induction on T.

For $T = []$, it is trivially $f_i(T, \overline{x}) = M_i^0(\overline{x}) = \| exp_i^0(\overline{x}) \|$, as $exp_i^0(\overline{x})$ contains no calls to other attribute functions. If $f_i(tl(T), \overline{x}) = M_i(\overline{x})$, $i \in 1..n$, we need to prove that:

$$f_i(T, \overline{x}) = next(M_i)(hd(T))(\overline{x}),\ i \in 1..n. \tag{1}$$

which we do again by induction on i.

Starting with $i = 1$, $next(M_i)(hd(T))(\overline{x})$, can be written as:

$$\| exp_1^{j,k}(\overline{x})[f_l \leftarrow \text{if } l < 1 \text{ then } next(M_l)(hd(T)) \text{ else } M_l] \|,$$

where k is specified by $hd(T)$ [5] and all calls to f_l, $l \in 2..n$ are replaced by M_l. Thus, due to the inductive hypothesis, it will be:

$$\| exp_1^{j,k}(\overline{x}) \| = \| exp_1^{j,k}(\overline{x})[f_l \leftarrow M_l] \|$$

A similar result holds for $cond_1^{j,k}$.

For $i > 1$, we rely on $f_l = next(M_l)(\rho_j)$, $l < i$, which guarantees that the property 1 holds for all values $l < i$.

This completes the proof of the case $(T_{T/M} - 10)$. □

Theorem 2. *$TS_{T/M}$ and TS_M are equivalent w.r.t. bisimulation.*

Proof. The proof is straightforward, because the effect of the trace on the attribute functions and the program execution is coded in memory M. Hence, intuitively the trace is redundant. □

Corollary 1. *TS_T and TS_M are equivalent w.r.t. bisimulation.*

Proof. Combining the two Theorems and the transitivity of bisimulation. □

4 Demonstration in LNT

The translation of EB^3 specifications is formalized in [18]. We show here how Sem_M facilitates the translation of EB^3 specifications to LNT for verification with the toolbox CADP. To this end, we present the translation of the EB^3 specification of Fig. 1 into LNT for $BID=\{b_1\}$ and $MID=\{m_1, m_2\}$ as was produced by the $EB^3 2LNT$ compiler [18].

LNT. LNT combines, in our opinion, features of imperative and functional programming languages and value-passing process algebras. It has a user-friendly syntax and formal operational semantics defined in terms of labeled transition

[5] see Fig. 5

$$B ::= \textbf{stop} \,\Big|\, \textbf{null} \,\Big|\, G(O_1, \ldots, O_n) \textbf{ where } E \,\Big|\, B_1; B_2 \,\Big|\, \textbf{if } E \textbf{ then } B_1 \textbf{ else } B_2 \textbf{ end if} \,\Big|$$
$$\textbf{var } x{:}T \textbf{ in } B \textbf{ end var} \,\Big|\, x := E \,\Big|\, \textbf{loop } L \textbf{ in } B \textbf{ end loop} \,\Big|\, \textbf{break } L \,\Big|$$
$$\textbf{select } B_1 [\,] \ldots [\,] B_n \textbf{ end select} \,\Big|\, \textbf{par } G_1, \ldots, G_n \textbf{ in } B_1 || \ldots || B_n \textbf{ end par} \,\Big|$$
$$P[G_1, \ldots, G_n](E_1, \ldots, E_n)$$
$$O ::= !E \,\Big|\, ?x$$

Fig. 7. Syntax of LNT

systems (LTSs). LNT is supported by the LNT.OPEN tool of CADP, which allows the on-the-fly exploration of the LTS corresponding to an LNT specification. We present the fragment of LNT that is useful for this translation. Its syntax is given in Fig. 7. LNT terms denoted by B are built from actions, choice (**select**), conditional (**if**), sequential composition (;), breakable loop (**loop** and **break**) and parallel composition (**par**). Communication is carried out by rendezvous on gates G with bidirectional transmission of multiple values. Gates in LNT (denoted with letter G with or without subscripts) correspond to the notion of labels in EB^3. Their parameters are called offers [6]. An offer O can be either a send offer (!) or a receive offer (?). Synchronizations may also contain optional guards (**where**) expressing boolean conditions on received values. The special action δ is used for defining the semantics of sequential composition. The internal action is denoted by the special gate i, which cannot be used for synchronization. The parallel composition operator allows multiway rendezvous on the same gate. Expressions E are built from variables, type constructors, function applications and constants. Labels L identify loops, which can be stopped using "**break**L" from inside the loop body. The last syntactic construct defines calls to process P that take gates $G_1, \ldots, G_n$ and variables $E_1, \ldots, E_n$ as actual parameters. The semantics of LNT are formally defined in [5].

Formalization. The principal gain from Sem_M lies in the use of *attribute variables*, the memory that keeps the values to all attribute functions. We need a mechanism that simulates this memory in LNT. The theoretical foundations of our approach are developed in [18]. In particular, we explicitly model in LNT a memory, which stores the *attribute variables* and is modified each time an action is executed. We model the memory as a process M placed in parallel with the rest of the system (a common approach in process algebra). To read the values of attribute variables, processes need to communicate with the memory M, and every action must have an immediate effect on the memory (so as to reflect the immediate effect on the execution trace). To achieve this, the memory process synchronizes with the rest of the system on every possible action of the system, and updates its attribute variables accordingly. Additional offers are used on each action, so that the current value of attribute variables can be read by processes during communication, and used to evaluate guarded expressions wherever needed.

[6] Offers are not explicitely mentioned in the syntactic rules for **par** and for procedural calls

```
process M [ACQ, DIS, REG, UNREG, LEND, RET : ANY] is
var mId : MEMBERID, bid : BOOKID, borrower : BOR, nbLoans : NB in
  (* attribute variables initialized *)
 mId := m_bot; borrower := BOR(m_bot); nbLoans := NB(0);
loop select
   ACQ (?bid) [] DIS (?bid, ?borrower)
[] REG (?mid) [] UNREG (?mid)
[] LEND (?bid, ?mid, !nbLoans, !borrower); borrower[ord (bid)] := mid;
   nbLoans[ord (mid)] := nbLoans[ord (mid)] + 1
[] RET (?bid); mId := borrower[ord (bid)]; borrower[ord (bid)] := m_bot;
   nbLoans[ord (mid)] := nbLoans[ord (mid)] - 1
end select end loop
end var end process
```

Fig. 8. Memory in LNT

These ideas are implemented in a tool called $EB^3$2LNT, presented in the companion paper [18]. We provide here the translation of the case study of library (with two members and two books) into LNT, obtained using $EB^3$2LNT.

Process M is given in Fig. 8. It runs an infinite loop, which "listens" to all possible actions of the system. We define two instances of the attribute variable *nbLoans* (one for each member) and one instance for *borrower* (one book). In the LNT expression *nbLoans*[**ord** (*mid*)], **ord** (*mid*) denotes the ordinate of value *mid*, i.e., a unique number between 0 and the cardinal of *mid*'s type minus 1. *nbLoans*[**ord** (*mid*)] is incremented after a *Lend* and decremented after a *Return* [7]. The action $Lend(mId, bId)$ takes, besides *mid* and *bid*, *nbLoans* and *borrower* as parameters, because the latter are used in the evaluation of the guarded expression preceding *Lend* (**where** statement in Fig. 8). Note how upon synchronisation on *Lend*, *nbLoans* and *borrower* are offered (!) by M and received (?) by *loan* (Fig. 8).

The main program is given in Fig. 9. All parallel quantification operations have been expanded as LNT is more structured and verbose than EB^3. For most EB^3 operators, there are equivalent LNT operators [18]. Making use of the expansion rule $E^* = \lambda \mid E.E^*$, the Kleene Closure (as in *member(mId)** in Fig. 1) can be written accordingly. The full LNT program is in the appendix.

5 Conclusion

In this paper, we presented an alternative, traceless semantics Sem_M for EB^3 that we proved equivalent to the standard semantics Sem_T. We showed how Sem_M facilitates the translation of EB^3 specifications to LNT for verification of temporal properties with CADP, by means of a translation in which the memory used to model attribute functions is implemented using an extra process that computes at each step the effect of each action on the memory. We presented the LNT translation of a case study involving a library with a predefined number

[7] see the definition of *nbLoans* in Fig. 1

```
process Main [ACQ, DIS, REG, UNREG, LEND, RET : ANY] () is
par ACQ, DIS, REG, UNREG, LEND, RET in
  par
     book [ACQ, DIS] (b1)
  ||
     par
        loop L in select break L [] member [REG, UNREG, LEND, RET] (m1)
        end select end loop
     ||
        loop L in select break L [] member [REG, UNREG, LEND, RET] (m2)
        end select end loop
     end par
  end par
||
  M [ACQ, DIS, REG, UNREG, LEND, RET]
end par
end process

process loan [LEND, RET : ANY](mid: MEMBERID, bid : BOOKID) is
var borrower: BOR, nbLoans: NB in  (* NbLoans is set to 1 *)
  LEND (bid, mid, ?nbLoans, ?borrower) where
     ((borrower[ord (bid)] eq m_bot) and (nbLoans[ord (mid)] eq 1));
  RET (bid)
end var
end process
```

Fig. 9. Main program and the process associated with the computation of the attribute function *Loan* in LNT

of books and members, translation obtained with the aid of a compiler called $EB^3$2LNT. The $EB^3$2LNT tool is presented in detail in [18].

A formal proof of the correctness of the $EB^3$2LNT compiler is under preparation. The proof strategy is by proving that the memory semantics of each EB^3 specification and its LNT translation are bisimilar, and works by providing a match between the reduction rules of Sem_M and the corresponding LNT rules [5].

As future work, we plan to study abstraction techniques for the verification of properties regardless of the number of components e.g. members, books that participate in the IS (Parameterized Model Checking). We will observe how the insertion of new functionalities to the ISs affects this issue. Finally, we will formalize this in the context of EB^3 specifications.

References

1. Abrial, J.-R.: The B-Book - Assigning programs to meanings. Cambridge University Press (2005)
2. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of Process Algebra. Elsevier (2001)

3. Bergstra, J.A., Klop, J.W.: Algebra of Communicating Processes with Abstraction. Journal of Theor. Comput. Sci. 37, 77–121 (1985)
4. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems 14(1), 25–59 (1987)
5. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator - Version 5.4. INRIA/VASY (2011)
6. Chossart, R.: Évaluation d'outils de vérification pour les spécifications de systèmes d'information. Master's thesis, Université de Sherbrooke (2010)
7. ClearSy. Atelier B, http://www.atelierb.societe.com
8. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation (extended abstract). In: Proc. of LICS, pp. 118–129 (1990)
9. Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., Ouenzar, M.: Comparison of model checking tools for information systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 581–596. Springer, Heidelberg (2010)
10. Frappier, M., St.-Denis, R.: EB^3: an entity-based black-box specification method for information systems. In: Proc. of Software and System Modeling (2003)
11. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A toolbox for the construction and analysis of distributed processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
12. Gervais, F.: Combinaison de spécifications formelles pour la modélisation des systèmes d'information. PhD thesis (2006)
13. Kozen, D.: Results on the propositional mu-calculus. Journal of Theor. Comput. Sci. 27, 333–354 (1983)
14. Löding, C., Serre, O.: Propositional dynamic logic with recursive programs. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 292–306. Springer, Heidelberg (2006)
15. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
16. Mousavi, M.R., Reniers, M.A.: Congruence for Structural Congruences. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 47–62. Springer, Heidelberg (2005)
17. Symons, V., Geoff, W.: The Evaluation of Information Systems: A Critique. Journal of Applied Systems Analysis 15 (1988)
18. Vekris, D., Lang, F., Dima, C., Mateescu, R.: Verification of EB^3 specifications using CADP, http://hal.inria.fr/hal-00768310

A LTS Construction

The construction is given by structural induction on E. In particular, we show how to construct:

$$TS_E = (S_E, \delta_E, I_E)$$

w.r.t. Sem_M for several cases of E. We refer to the initial memory as $M^0 \in \mathcal{M}$ ($\mathcal{M}$ is the set of memory mappings in the IS) defined upon the fixed body of attribute function definitions. It is $I_E = \{(E, M^0)\}$. More precisely:

$$S_\rho = \{(\rho, M^0)\} \bigcup \{(\surd, next(M^0))\},\ \delta_\rho = \{(\rho, M^0) \xrightarrow{\rho} (\surd, next(M^0))\},\ \rho \neq \lambda$$
$$S_\lambda = \{(\lambda, M^0)\} \bigcup \{(\surd, M^0)\},\ \delta_\lambda = \{(\lambda, M^0) \xrightarrow{\lambda} (\surd, M^0)\}$$
$$S_{E_1.E_2} = \{(E_1'.E_2, M) \mid (E_1', M) \in S_{E_1}\} \bigcup \{\bigcup_{(\surd, M_1) \in S_{E_1}} S_{E_2}^{M_1}\}$$
$$\delta_{E_1.E_2} = \{(E_1'.E_2, M) \xrightarrow{\rho} (E_1''.E_2, M') \mid (E_1', M) \xrightarrow{\rho} (E_1'', M') \in \delta_{E_1}\} \bigcup \{\bigcup_{(\surd, M_1) \in S_{E_1}} \delta_{E_2}^{M_1}\}$$
$$TS_{E^*} = lfp_F,\ \text{where } F(TS_{E_x}) = TS_{E \cdot E_x} \cup TS_\lambda, \quad E^* = \lambda \mid E.E^*$$
$$TS_{E_1|[\Delta]|E_2},\ \text{where } E_1|[\Delta]|E_2 \doteq \textstyle\sum_{i=1}^r C\,(T)^i \Rightarrow \rho^i(\overline{a}^i).E^i$$
$$S_{C\,(T) \Rightarrow E} = \begin{cases} \{(C\,(T) \Rightarrow E, M^0)\} \bigcup S_E \setminus \{(E, M^0)\}, & \text{if } \|C[f_i \leftarrow M_i^0]\| \\ \{(C\,(T) \Rightarrow E, M^0)\}, & \text{otherwise} \end{cases}$$
$$\delta_{C\,(T) \Rightarrow E} = \begin{cases} \delta_E[(E, M^0) \leftarrow (C\,(T) \Rightarrow E, M^0)], & \text{if } \|C[f_i \leftarrow M_i^0]\| \\ \emptyset, & \text{otherwise} \end{cases}$$

For $(E_1', M) \in S_{E_1}$, it will be $(E_1'.E_2, M) \in S_{E_1 \cdot E_2}$. For $(\surd, M_1) \in S_{E_1}$, we obtain $(E_2', M) \in S_{E_2}^{M_1}$, where $S_{E_2}^{M_1}$ stands for state space S_{E_2} with initial memory M_1. For TS_{E^*}, we need to compute the least fix point of function $F : TS \rightarrow TS$ w.r.t. the *lattice* $\mathcal{TS} = (TS, \subseteq)$, where TS is the possibly infinite set of LTSs simulating EB^3 specifications w.r.t. $Sem_{T/M}$ and $\subseteq$ denotes inclusion. For $TS_{E_1|[\Delta]|E_2}$, $E_1|[\Delta]|E_2$ is written as a sum of EB^3 expressions. The first action of each summand would be $\rho^i(\overline{a}^i)$ for all possible execution paths picking this summand. This action would be taken under C^i (=*true* in the absence of condition):

$$E_1|[\Delta]|E_2 \doteq \sum_{i=1}^{r} C\,(T)^i \Rightarrow \rho^i(\overline{a}^i).E^i$$

This form is known as *head normal form* (HNF) in the literature. The construction of HNFs for process algebra expressions is discussed in [2]. It is a common practice developed principally in the context of the Algebra of Communicating Processes (ACP) [3] as a means to analyse the behaviour of recursive process algebra definitions. Note that the rules $(S_M - 8)$ and $(S_M - 9)$ for Sem_M ensure the existence of this normal form. In the last case, for $\|C[f_i \leftarrow M_i^0]\| = true$, we need to construct S_E and replace (E, M^0) with $(C\,(T) \Rightarrow E, M^0)$.

B LNT code for the Library Management System

```
module Libr_Manag_Syst is
type MEMBERID is m1, m2, m_bot with "eq", "ne", "ord" end type
type BOOKID is b1, b_bot with "eq", "ne", "ord" end type
type ACQUIR is array [0..1] of BOOL end type
type NB is array [0..2] of NAT end type
type BOR is array [0..1] of MEMBERID end type

process M [ACQ, DIS, REG, UNREG, LEND, RET : ANY] is
var mId : MEMBERID, bid : BOOKID, borrower : BOR, nbLoans : NB in
  (* attribute variables initialized *)
```

```
  mId := m_bot; borrower := BOR(m_bot); nbLoans := NB(0);
loop select
   ACQ (?bid) [] DIS (?bid, ?borrower)
[] REG (?mid) [] UNREG (?mid)
[] LEND (?bid, ?mid, !nbLoans, !borrower); borrower[ord (bid)] := mid;
   nbLoans[ord (mid)] := nbLoans[ord (mid)] + 1
[] RET (?bid); mId := borrower[ord (bid)]; borrower[ord (bid)] := m_bot;
   nbLoans[ord (mid)] := nbLoans[ord (mid)] - 1
end select end loop
end var end process

process loan [LEND, RET : ANY] (mid : MEMBERID, bid : BOOKID) is
var borrower : BOR, nbLoans : NB in  (* NbLoans is set to 1 *)
  LEND (bid, mid, ?nbLoans, ?borrower) where
     ((borrower[ord (bid)] eq m_bot) and (nbLoans[ord (mid)] eq 1));
  RET (bid)
end var end process

process book [ACQ, DIS : ANY] (bid : BOOKID) is
var borrower: BOR in
  ACQ (bid); DIS (bid, ?borrower) where (borrower[ord (bid)] eq m_bot)
end var end process

process member [REG, UNREG, LEND, RET : ANY] (mid : MEMBERID) is
  REG (mid);
  loop L in select break L [] loan [LEND, RET] (mid, b1)
            end select end loop; UNREG (mid)
end process

process Main [ACQ, DIS, REG, UNREG, LEND, RET : ANY] () is
par ACQ, DIS, REG, UNREG, LEND, RET in
  par
     book [ACQ, DIS] (b1)
  ||
     par
        loop L in select break L [] member [REG, UNREG, LEND, RET] (m1)
        end select end loop
     ||
        loop L in select break L [] member [REG, UNREG, LEND, RET] (m2)
        end select end loop
     end par
  end par
|| M [ACQ, DIS, REG, UNREG, LEND, RET]
end par
end process
end module
```

Interval Soundness of Resource-Constrained Workflow Nets: Decidability and Repair

Elham Ramezani, Natalia Sidorova, and Christian Stahl

Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven,
The Netherlands
{E.Ramezani,N.Sidorova,C.Stahl}@tue.nl

Abstract. Correctness of workflow design cannot be evaluated by checking the execution for one single instance of the workflow, because instances, even when being independent from the data perspective, depend on each other with respect to the resources they rely on for executing tasks. The resources are *shared among the instances* of the same workflow; moreover, other workflows can use the same resources. Therefore, we enrich the workflow model with the model of its *environment* that captures the resource perspective. This allows us to investigate the verification of workflows extended with resources in a more general setting than it was previously done. We focus on the *soundness* property, which means the ability to terminate properly from any reachable state of the system, for every instance of the system. We show the *decision* procedure for soundness and how to *repair* a workflow that is unsound from the resource perspective by synthesizing a controller such that the composition of the workflow and the controller is sound by design.

1 Introduction

A workflow consists of a set of coordinated tasks describing a flow of work for accomplishing some business process within an organization. The occurrence of those tasks may depend on *resources*, such as machines, manpower, and raw material. Often, several *cases* (i.e., instances) of a workflow coexist, and they may all concurrently access the resources. In that sense, the execution of a workflow is similar to executing several threads of a piece of software.

Correctness of classical and resource-constrained workflows has been formalized in terms of the *soundness* property [1,14]. Soundness guarantees that given a finite number of cases and a number of resources of each type, every case has always the possibility to terminate. As we restrict ourselves to *durable* resources in this paper—that is, resources that can neither be created nor destroyed—soundness also ensures that the number of resources initially available remains invariant.

The current notion of soundness for resource-constrained workflows assumes a workflow to be executed in isolation. However, workflows increasingly cross organizational boundaries and are usually intertwined. As a consequence, resources

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 150–167, 2013.
DOI: 10.1007/978-3-642-40213-5_10,
© IFIP International Federation for Information Processing 2013

are no longer internal for a workflow but shared among different workflows. This, in fact, requires a different way of modeling workflows. To do so, we propose to enrich the workflow model with an *environment capturing the resource perspective of the workflow*. The environment is generic in the sense that it can be parameterized, thereby enabling the modeling of relevant instances of practical scenarios. More precisely, the environment allows for borrowing, lending, and permanently adding and removing of resources of each type up to an initially specified number. Moreover, it also creates the cases of the workflow that are to be executed, with the number of cases taken from a specified interval.

We formalize correctness of workflows with shared resources with the notion of *interval soundness*, as it considers intervals of cases and resources. Interval soundness is defined for the composition of the workflow and the corresponding instance of the generic environment. We show that the verification of soundness reduces to check whether for the workflow it is always possible to terminate in the composition with the environment. The state of the environment can thereby be neglected because several invariant properties hold in the environment model which are necessary for interval soundness. To further support the design of interval-sound workflows, we present an approach for *repairing* an unsound workflow by synthesizing a controller (if exists) such that the composition of the workflow and the controller is interval sound.

Our contributions can be summarized as follows:

– A generalization of the model for workflows extended with resources to deal with shared resources;
– A notion of correctness considering intervals of instances and resource vectors and two procedures to decide correctness; and
– An approach to repair an incorrect workflow based on controller synthesis.

We continue by providing the background in Sect. 2. In Sect. 3, we introduce our model of resource-constrained workflow nets, the generic environment for modeling the resource perspective, and define interval soundness. In Sect. 4, we show how interval soundness can be decided, and repairing unsound workflows is studied in Sect. 5. We discuss related work in Sect. 6 and close with a conclusion.

2 Preliminaries

In this section, we provide the basic notations used in this paper, such as Petri nets and workflow nets.

For two sets P and Q, let $P \uplus Q$ denote the disjoint union; writing $P \uplus Q$ expresses the implicit assumption that P and Q are disjoint. A *multiset* or bag m over P is a mapping $m : P \to \mathbb{N}$; for example, $[p_1, 2p_2]$ denotes a multiset m with $m(p_1) = 1$, $m(p_2) = 2$, and $m(p) = 0$ for $p \in P \setminus \{p_1, p_2\}$. We define operations $+, -, =, <, >, \leq, \geq$ on multisets in the standard way. We overload the set notation, writing $\emptyset$ for the empty multiset and $\in$ for the element inclusion. We canonically extend the notion of a multiset over P to supersets $Q \supseteq P$; that is, for a mapping $m : P \to \mathbb{N}$, we extend m to the multiset $m : Q \to \mathbb{N}$ so that

for all $p \in Q \setminus P$, $m(p) = 0$. Analogously, a multiset can be restricted to a subset $Q \subseteq P$. For a mapping $m : P \to \mathbb{N}$, the *restriction* of m to the elements in Q is denoted by $m|_Q : Q \to \mathbb{N}$.

Definition 1. (labeled Petri net) A *net* $N = \langle P, T, W \rangle$ consists of

- a finite set P of *places*,
- a finite set T of *transitions* such that P and T are disjoint, and
- a *weight function* $W : (P \times T) \uplus (T \times P) \to \mathbb{N}$.

A *labeled net* $N = \langle P, T, W, l, \Sigma \rangle$ is a net $\langle P, T, W \rangle$ together with an *alphabet* Σ of *actions* and a *labeling function* $l : T \to \Sigma \uplus \{\tau\}$, where τ represents an invisible, internal action. A *(labeled) Petri net* $\langle N, m_N \rangle$ is a (labeled) net N together with an *initial marking* m_N, where a *marking* $m : P \to \mathbb{N}$ is a distribution of tokens over the places. The *incidence matrix* $\mathbf{C}$ of N is defined by $\forall (p,t) \in P \times T : \mathbf{C}(p,t) = W((t,p)) - W((p,t))$.

For a transition $t \in T$, we define the *preset* ${}^\bullet t$ and the *postset* $t^\bullet$ of t as the multisets of places where every $p \in P$ occurs $W((p,t))$ times in ${}^\bullet t$ and $W((t,p))$ times in $t^\bullet$. Analogously, we define for a place $p \in P$ its preset ${}^\bullet p$ and its postset $p^\bullet$. We also lift pre- and postsets to sets of places and of transitions. A place p is a *source* place if ${}^\bullet p = \emptyset$ and a *sink* place if $p^\bullet = \emptyset$.

A transition $t \in T$ is *enabled* at a marking m, denoted by $m \xrightarrow{t}$, if ${}^\bullet t \leq m$. If t is enabled at m, it can *fire*, thereby changing the marking m to a marking $m' = m - {}^\bullet t + t^\bullet$. The firing of t is denoted by $m \xrightarrow{t} m'$; that is, t is enabled at m and firing t results in m'. Depending on the context, we interpret a marking m of N either as a multiset over P or as a vector from $P \to \mathbb{N}$. Firing transitions can be extended to sequences: $m_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-1}} m_k$ is a *run* of N if for all $0 < i < k$, $m_i \xrightarrow{t_i} m_{i+1}$. A marking m' is *reachable from* a marking m if there exists a (possibly empty) run $m_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-1}} m_k$ with $m = m_1$ and $m' = m_k$; for $v = t_1 \ldots t_{k-1}$, we also write $m \xrightarrow{v} m'$. Marking m' is *reachable* if $m_N = m$. The set $\mathcal{R}(m)$ represents all markings of N that are reachable from m.

A marking m of N is *b-bounded* for a bound $b \in \mathbb{N}$, if $m(p) \leq b$ for all $p \in P$. N is bounded if every reachable marking is b-bounded for some $b \in \mathbb{N}$. A transition $t \in T$ is *live* if from every reachable marking m there is a marking m' such that t is enabled at m'. If all transitions are live, then N is live. A marking m is a *home-marking* if from every reachable marking we can reach m. A set *HS* of markings of N is a *home-space* if for every reachable marking m, there exists a marking $m' \in HS$ such that m' is reachable from m.

A *place invariant* is a row vector $I : P \to \mathbb{Q}$ such that $I \cdot \mathbf{C} = 0$. When talking about invariants, we consider markings as *vectors*.

In the following, we define two composition operators for labeled Petri nets to model asynchronous composition based on place fusion and synchronous parallel composition based on transition fusion. The composition operator $\oplus$ merges common places of two labeled Petri nets.

Definition 2. (asynchronous composition) Two labeled nets N_1 and N_2 are *a-composable* if $(\Sigma_1 \cup T_1) \cap (\Sigma_2 \cup T_2) = \emptyset$. The *asynchronous composition* of two a-composable labeled nets is the labeled net $N_1 \oplus N_2 = \langle P_1 \cup P_2, T_1 \uplus T_2, W_1 \uplus W_2, l, \Sigma_1 \uplus \Sigma_2 \rangle$ and $l(t) = l_i(t)$ for $t \in T_i$, $i = 1, 2$.

If N_1 and N_2 are labeled Petri nets with initial markings m_{N_1} and m_{N_2}, then the composition is a labeled Petri net with initial marking $m_0 = m_{N_1} + m_{N_2}$.

We define a synchronous composition operator $\|$ where, for each common action a, an a-labeled transition of one labeled Petri net is merged with an a-labeled transition of the other. If there is more than one a-labeled transition in one of the labeled Petri nets, then each of these transitions is merged with a copy of the respective transition of the other labeled Petri net.

Definition 3. (synchronous composition) Two labeled nets N_1 and N_2 are *s-composable* if $(P_1 \uplus T_1) \cap (P_2 \uplus T_2) = (\Sigma_1 \cap \Sigma_2)$. The *synchronous composition* of two s-composable labeled nets is the labeled net $N_1 \| N_2 = \langle P, T, W, l, \Sigma \rangle$, where

- $P = P_1 \uplus P_2$,
- $T = \quad \{t \in T_1 \cup T_2 \mid l_1(t) = \tau \vee l_2(t) = \tau\}, \uplus \{(t_1, t_2) \in T_1 \times T_2 \mid l_1(t_1) = l_2(t_2) \wedge l_1(t_1) \neq \tau\}$,
- $W((p,t)) = \begin{cases} W_i((p,t)), & p \in P_i, t \in T_i, l_i(t) = \tau, i = 1,2, \\ W_i((p,t_i)), & p \in P_i, t = (t_1,t_2), i = 1,2, \\ 0, & \text{otherwise}, \end{cases}$
  $W((t,p)) = \begin{cases} W_i((t,p)), & p \in P_i, t \in T_i, l_i(t) = \tau, i = 1,2, \\ W_i((t_i,p)), & p \in P_i, t = (t_1,t_2), i = 1,2, \\ 0, & \text{otherwise} \end{cases}$
- $l(t) = \begin{cases} l_i(t), & t \in T_i, i = 1,2, \\ l_1(t_1), & t = (t_1,t_2), \end{cases}$
- $\Sigma = \Sigma_1 \cup \Sigma_2$.

If N_1 and N_2 are labeled Petri nets with initial markings m_{N_1} and m_{N_2}, then composition yields a labeled Petri net with initial marking $m_0 = m_{N_1} + m_{N_2}$.

The *labeled transition system* (LTS) $TS_N = \langle Q, \delta, \hat{q}, \Sigma \rangle$ of a labeled Petri net $N = \langle P, T, W, l, \Sigma, m_N \rangle$ consists of a set $Q = \mathcal{R}(m_N)$ of *states*, a set δ of labeled *edges* with $(q, l(t), q') \in \delta$ iff $q \xrightarrow{t} q'$ and $q, q' \in Q$, and an *initial state* $\hat{q} = m_N$.

We define the synchronous product of two labeled transition systems in the standard way: common visible actions are synchronized, all other actions are not. In fact, we have $TS_{N_1 \| N_2}$ and $TS_{N_1} \| TS_{N_2}$ are isomorph.

Definition 4. (synchronous product) The synchronous product of two LTSs TS_1 and TS_2 is the LTS $TS_1 \| TS_2 = \langle Q_1 \times Q_2, \delta, (\hat{q}_1, \hat{q}_2), \Sigma_1 \cup \Sigma_2 \rangle$ with

$$\begin{aligned} \delta = \quad & \{((q_1,q_2), x, (q_1',q_2')) \mid (q_1, x, q_1') \in \delta_1, (q_2, x, q_2') \in \delta_2, x \in \Sigma_1 \cup \Sigma_2\} \\ \uplus\, & \{((q_1,q_2), \tau, (q_1',q_2)) \mid (q_1, \tau, q_1') \in \delta_1\} \\ \uplus\, & \{((q_1,q_2), \tau, (q_1,q_2')) \mid (q_2, \tau, q_2') \in \delta_2\}. \end{aligned}$$

Workflow Nets A workflow refers to the automation of processes by an IT infrastructure, in whole or in part [3]. Workflows are *case*-based; that is, every piece of work is executed for a specific case. The workflow definition specifies which tasks need to be executed for a case and in what order.

We can model a workflow definition as a (labeled) net, thereby modeling tasks by transitions and conditions by places; the state of a case is captured by a marking of the net. The assumption that a typical workflow has a well-defined starting point and a well-defined ending point imposes syntactic restrictions on Petri nets that result in the following definition of a workflow net [2].

Definition 5. (WF-net) A labeled net $N = \langle P, T, W, l, \Sigma \rangle$ is a *workflow net* (WF-net) if it has a nonempty set of transitions, a single source place i, a single sink place f, and every place and every transition is on a path from i to f.

The *short-circuited net* N_s of N is the labeled net obtained from N by adding a transition t_s with $W((t,i)) = W((f,t)) = 1$ and $l(t_s) = \tau$.

In the first instance, researchers were interested in workflow correctness with respect to a single case. One of the most established correctness properties of WF-nets is *soundness*, as introduced by Van der Aalst [1] in the context of one case. Soundness guarantees that the workflow has always the possibility to terminate. Later on, multi-instance behavior attracted researchers' attention, where WF-nets are considered as parameterized systems modeling the processing of batches of tasks, as introduced in [14]. While in classical workflows cases are considered to be independent and the modeling of multiple cases in one WF-net requires the introduction of id tokens, in batch workflows cases are considered to be undistinguishable and mixable (e.g., it does not matter which employee works on which order) and, as a consequence, cases are modeled with undistinguishable black tokens. Under certain conditions on the workflow structure, called *separability*, the behavior of the WF-net with undistinguishable cases (black tokens) is equivalent (up to trace equivalence) to the behavior of the WF-net with id tokens [14,8,7]. Moreover, every net with id tokens can be transformed into an up-to-bisimulation-equivalent net with black tokens only [14,17].

Capturing the correctness notion for batch workflow nets requires the use of the generalized notion of soundness, as proposed in [14].

Definition 6. (WF-net soundness) Let $k \in \mathbb{N}$. A WF-net N is *k-sound* if, for every marking m reachable from marking $[k \cdot i]$, we can reach marking $[k \cdot f]$.

The next definition gives a requirement for the correct design of a workflow that can be checked using structural properties of the net [15]. *Nonredundancy* of a place $p \in P$ guarantees that p can potentially be marked with a token in some reachable marking.

Definition 7. Let $N = \langle P, T, W, l, \Sigma \rangle$ be a WF-net. A place $p \in P$ is *nonredundant* if there exist $k \in \mathbb{N}$ and $m \in \mathbb{N}^P$ such that $[k \cdot i] \xrightarrow{*} m \wedge p \in m$.

3 Generalizing Resource-Constrained Workflow Nets

We use the notion of resource-constrained workflow nets (RCWF-nets) [16] to extend the definition of the workflow with resource dependencies of the tasks. The production net of an RCWF-net is a WF-net in its traditional sense, defining the order of task execution, resource places model the resource types used by the workflow, and resource consumption and production are modeled by the arcs from the resource places to the transitions of the production net, and vice versa.

Definition 8. (RCWF-net) A labeled net $N = \langle P_p \uplus P_r, T, W_p \uplus W_r, l, \Sigma \rangle$ is a *resource-constrained workflow net* (RCWF-net) if

- $N_p = \langle P_p, T, W_p, l, \Sigma \rangle$ is a WF-net, the *production net* of N;
- P_p is the set of *production places*, and P_r is the set of *resource places*; and
- $W_r : (P_r \times T) \cup (T \times P_r) \to \mathbb{N}$ is the *resource weight function*.

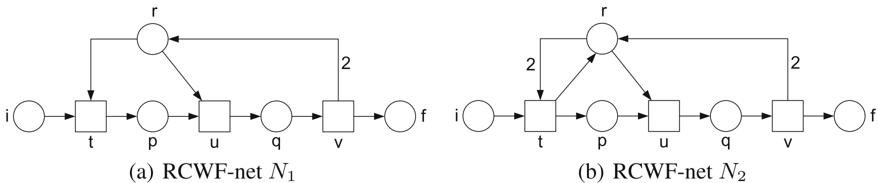
The *short-circuited net* N_s of N is the labeled net obtained from N by replacing N_p with its short-circuit net.

The initial marking $m_N = [k \cdot i] + R$ of an RCWF-net N consists of $k \in \mathbb{N}$ tokens in place i, specifying the number of cases in the workflow that are concurrently executed, and an initial marking for the set P_r of resources places, denoted as a resource vector $R \in \mathbb{N}^{P_r}$.

Example 1. We illustrate the previously introduced concepts using Fig. 1. The nets N_1 and N_2 are RCWF-nets with one resource place r. Arc weights are depicted on the respective arc unless they are equal to 1. Erasing r and its adjacent arcs from N_1 and N_2 yields the (same) production net, a WF-net. This WF-net is k-sound, for any $k > 0$.

3.1 The Generic Environment

To consider RCWF-nets in the setting where the workflow works within some environment that can borrow resources from the workflow or lend more resources to it, we introduce patterns capturing typical behavior of the resource environment. We consider the following actions of the environment: *borrowing* resources (the borrowed resources are then used by other workflows and they can eventually be returned and made available for the workflow again), *lending* resources (i.e.,



(a) RCWF-net N_1 (b) RCWF-net N_2

Fig. 1. Example of an unsound and a sound RCWF-net

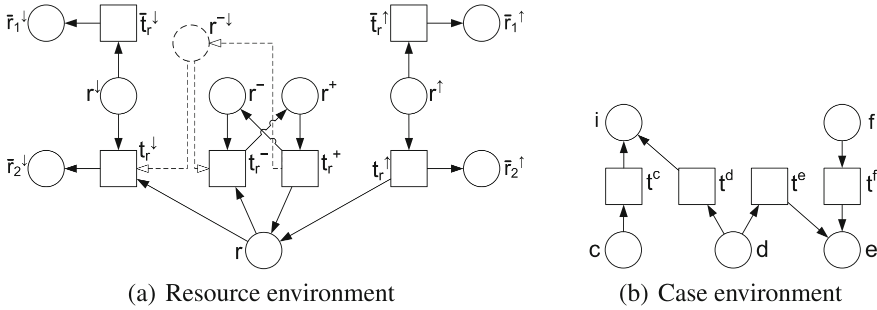


Fig. 2. Generic resource environment for the resource place r and case environment for a workflow with initial place i and final place f

making some additional resources temporarily available, and eventually taking them back, when unused by the workflow), *permanently removing* resources, and *permanently adding* resources. Actual environments allow for a (possibly empty) subset of these actions.

We define the generic environment, built as a union of generic environments defined for every resource type (i.e., place). All the patterns can be obtained by choosing an appropriate initial marking for the generic environment. The generic environment of a resource place r is captured in Fig. 2(a). The transitions $t_r^{\uparrow}$ and $t_r^{\downarrow}$ model permanent addition and removal of resources correspondingly. Their counterparts $\overline{t}_r^{\uparrow}$ and $\overline{t}_r^{\downarrow}$ model the decision of the environment not to lend/borrow a certain number of resources, but removing the tokens from places $r^{\uparrow}$ and $r^{\downarrow}$. The number of tokens in places $r^{\uparrow}$ and $r^{\downarrow}$ in the initial marking gives the bounds for the number of resources that can be added and removed, respectively. Clearly, choosing 0 as initial marking of $r^{\uparrow}$ and $r^{\downarrow}$ makes the corresponding transitions dead, so they might be removed for the corresponding pattern, but for the sake of readability, we prefer to use only the initial marking for configuring the generic environment into a pattern.

Transitions t_r^{+} and t_r^{-} together with places r^{+} and r^{-} model lending and borrowing resources in the following way: The number of tokens in r^{+} and r^{-} in the initial marking corresponds to the number of resources the environment may lend and borrow, respectively. Thus having 0 for the initial marking on r^{+} means that the environment cannot lend any resource of type r. When the resources are borrowed, the marking of r^{+} is increased by the number of borrowed resources, and the firings of t_r^{+} will then correspond to returning those resources by the environment.

The initial marking of the places $r^{\uparrow}$, $r^{\downarrow}$, r^{+}, and r^{-} serves thus as the configuration parameter defining the behavior of the environment. In principle, it is possible to elaborate the model further by linking, for example, $r^{\downarrow}$ and r^{-} by means of choosing the bound for the total number of resources that can be removed permanently or temporarily. We can model this by introducing the place $r^{-\downarrow}$ and the arcs depicted by the dashed lines in Fig. 2(a). The same can be done

for other configuration components. Variations on the environment construction are also possible by linking the scheme for adding and removing resources for different resource types. We restrict our attention further to the main structure, without linking the configuration components to each other, although the results hold for environments restricted in this way, too.

Since borrowing is temporary—that is, under the fairness assumption, the environment will eventually return the borrowed resources—the choice of the initial marking for r^- does not change the set of markings reachable in the composition of the workflow and the environment projected on the workflow places: The workflow can always wait until the environment returns the resources borrowed and then proceed. The same applies to lending resources: The workflow can always wait until the environment will lend it the maximal amount of the resources possible, meaning that the behavior of the composition is defined by r^+. The borrowing/lending part of the environment model becomes important when time is taken into consideration, also for transitions t_r^+ and t_r^-, since borrowing/lending then changes the set of markings reachable in the workflow. Note that also transition $\bar{t}_r^{\downarrow}$, decreasing the amount of resources the environment might remove permanently, only has influence on the set of markings reachable in the workflow when we take time into account.

To create cases of the workflow, we add a generic *case environment* to our generic environment, allowing for an arbitrary number of cases from the interval $[k_1, k_2]$, for some $k_1, k_2 \in \mathbb{N}$, $k_1 \leq k_2$. Figure 2(b) shows the construction. The place c (creation) contains initially k_1 tokens (i.e., the lower bound of cases to be created), the place d (dismissable) contains $k_2 - k_1$ tokens (i.e., cases that can but do not have to be created), and place e (end) is empty. For every case that is not created, a token is produced in the place e by firing t^e. Thus, if all created cases terminate (modeled by a token in f for each case), the place e contains on the termination k_2 tokens.

Definition 9. (generic environment) Let $N = \langle P_p \uplus P_r, T, W_p \uplus W_r, l, \Sigma\rangle$ be an RCWF-net. The *generic environment* of N is a labeled Petri net E such that E and N are a-composable with $((P_p \uplus P_r) \cap P_E) = P_r \uplus \{i, f\}$ and $E = \langle P_E, T_E, W, m_E, l_E, \{\tau\}\rangle$ is defined as

- $P_E = P_r \uplus P_e \uplus \{i, f, c, d, e\}$ with $P_e = \{r^-, r^+, r^{\uparrow}, r^{\downarrow}, \bar{r}_1^{\uparrow}, \bar{r}_2^{\uparrow}, \bar{r}_1^{\downarrow}, \bar{r}_2^{\downarrow} \mid r \in P_r\}$,
- $T_E = \{t_r^-, t_r^+, t^{\uparrow}, t^{\downarrow}, \bar{t}_r^{\uparrow}, \bar{t}_r^{\downarrow} \mid r \in P_r\} \uplus \{t^c, t^d, t^e, t^f\}$,
- $W((r^-, t_r^-)) = W((r, t_r^-)) = W((t_r^-, r^+)) = W((t_r^+, r^-)) = W((t_r^+, r)) = W((r^+, t_r^+)) = W((r^{\uparrow}, t_r^{\uparrow})) = W((t_r^{\uparrow}, \bar{r}_2^{\uparrow})) = W((t_r^{\uparrow}, r)) = W((r^{\uparrow}, \bar{t}_r^{\uparrow})) = W((\bar{t}_r^{\uparrow}, \bar{r}_1^{\uparrow},)) = W((r^{\downarrow}, t_r^{\downarrow})) = W((t_r^{\downarrow}, \bar{r}_2^{\downarrow})) = W((r, t_r^{\downarrow})) = W((r^{\downarrow}, \bar{t}_r^{\downarrow})) = W((\bar{t}_r^{\downarrow}, \bar{r}_1^{\downarrow},)) = 1$ for $r \in P_r$ and
  $W((c, t^c)) = W((t^c, i)) = W((d, t^d)) = W((d, t^e)) = W((t^d, i)) = W((t^e, e)) = W((f, t^f)) = W((t^f, e)) = 1$,

– $m_E(p) = \begin{cases} m_N(p), & p \in P_r \\ m_r^-, & p = r^- \textit{ and } m_r^- \textit{ is the maximal number of} \\ & \textit{resources } r \textit{ the environment can } \text{borrow} \\ m_r^+, & p = r^+ \textit{ and } m_r^+ \textit{ is the maximal number of} \\ & \textit{resources } r \textit{ the environment can } \text{lend} \\ m_r^\downarrow, & p = r^\downarrow \textit{ and } m_r^\downarrow \textit{ is the maximal number} \\ & \textit{resources } r \textit{ the environment can } \text{remove} \\ m_r^\uparrow, & p = r^\uparrow \textit{ and } m_r^\uparrow \textit{ is the maximal number of} \\ & \textit{resources } r \textit{ the environment can } \text{add} \\ k_1 \in \mathbb{N}, & p = c \\ k_2 - k_1 \in \mathbb{N}, & p = d \end{cases}$

An *environment* $\langle E, m_E \rangle$ of N consists of E and a concrete initial marking m_E.

3.2 Interval Soundness for RCWF-nets with an Environment

We adapt the definition of soundness for WF-nets to RCWF-nets with an environment. Soundness of an RCWF-net N with an environment $\langle E, m_E \rangle$ guarantees that the underlying production net of N is k-sound, for every k in the interval; that is, also in the presence of resources, a case has always the possibility to terminate. In addition, we put two conditions on the resources: First, all resources that are initially available in N and E are again available when all cases are terminated. Second, at any reachable marking, the number of available resources does not increase the number of initially available resources. These two criteria are a consequence of our restriction to durable resources, because they ensure that no resources are created or removed.

To guarantee the previous conditions, we define four necessary conditions that are captured in the notion of a *well-defined composition* of N and an arbitrary environment $\langle E, m_E \rangle$. The first condition ensures that the production net of N is k-sound, for every k in the interval. The second condition ensures that no resource tokens can be created by the WF-net; that is, for every firing sequence, the number of resource tokens put by N on the resource places does not exceed the number of tokens taken by N from the resource places (meaning that then every reachable marking has a resource vector $R' \leq R$, unless the environment can add tokens to the resource places). The third condition states that there exists a place invariant for the places c, d and e, guaranteeing that the number of cases remains constant. Likewise, the fourth condition requires that, for every resource place, there exists a place invariant, guaranteeing that the number of resources remains constant.

Definition 10. (well-defined) Let N be an RCWF-net such that the production net of N does not have redundant places. Let $\langle E, m_E \rangle$ be an environment of N. The composition $N \oplus E$ is *well-defined* if the following four properties hold:

1. The production net of N is k-sound, for all $m_E(c) \leq k \leq m_E(c) + m_E(d)$.
2. $\forall x \in \mathbb{Z}^T : (\mathbf{C} \cdot x)|_{P_p \uplus \{e\}} \geq 0$ implies $(\mathbf{C} \cdot x)|_{P_r} \leq 0$.
3. There exists a place invariant I_p such that $I_p(c) = I_p(d) = I_p(e) = 1$ and, for all $p' \in P_E \setminus \{c, d, e\}$, $I_p(p') = 0$.
4. For each $r \in P_r$, there exists a place invariant I_r satisfying $I_r(c) = I_r(d) = I_r(e) = 0, I_r(r) = 1$, and $\forall r' \in P_r \setminus \{r\} : I_r(r') = 0$.

The absence of redundant places is necessary for applying invariant techniques. The next lemma shows that a well-defined composition is bounded.

Lemma 11. *Let N be an RCWF-net and $\langle E, m_E \rangle$ be an environment of N. If $N \oplus E$ is well-defined, then it is bounded.*

Proof. Boundedness of the resource environment follows from Definitions 10(2), (4) and of the case environment from Definition 10(3). The latter argument and Definition 10(1), which implies boundedness of the production net, implies boundedness of N. □

For a well-defined composition $N \oplus E$, we can define interval soundness, which is a more general variant of the soundness notion as defined in [13,5].

Definition 12. (interval soundness) Let N be an RCWF-net and $\langle E, m_E \rangle$ be an environment of N such that $N \oplus E$ is well-defined. Then, N is *sound with* $\langle E, m_E \rangle$ if for all $m \in \mathcal{R}(m_{N \oplus E}) : m \xrightarrow{*} m'$ such that $m'(e) = m_E(c) + m_E(d)$. If m_E is not relevant, we say N is *interval sound*.

Definition 12 captures at least the following relevant instances of interval soundness:

– (k, R)-soundness [13,5] (i.e., we consider a fixed number k of cases and a fixed resource vector R) if $m_E(p) = 0$, for all $p \in P_e \uplus \{d\}$ and $m_E(r) = R(r)$, for all $r \in P_r$;
– up-to $(k, [R, R^+])$-soundness (i.e., (k, R')-soundness for all $R \leq R' \leq R^+$ but the initial resource vector R can be increased up to R^+ at runtime) if $m_E(d) = 0$ and $m_E(r^\downarrow) = 0$, for all $r \in P_r$ and $m_E(r^\uparrow) = R^+(r) - R(r)$, $m_E(r) = R(r)$ for all $r \in P_r$;
– down-to $(k, [R^-, R])$-soundness (i.e., (k, R')-soundness for all $R^- \leq R' \leq R$ but the initial resource vector R can be reduced down to R^- at runtime) if $m_E(d) = 0$ and $m_E(r^\uparrow) = 0$, for all $r \in P_r$ and $m_E(r^\downarrow) = R(r) - R^-(r)$, $m_E(r) = R(r)$ for all $r \in P_r$;
– up-to, down-to $(k, [R^-, R^+])$-soundness (i.e., (k, R)-soundness for all $R^- \leq R \leq R^+$ but the initial resource vector R can be reduced down to R^- or increased up to R^+ at runtime) if $m_E(d) = 0$ and $m_E(r^\uparrow) = R^+(r) - R(r)$, $m_E(r^\downarrow) = R(r) - R^-(r)$, $m_E(r) = R(r)$.

We now relate the previous variants of interval soundness, thereby generalizing them from a fixed number k of cases to an interval $[k_1, k_2]$ of cases.

Lemma 13. *For any RCWF-net N and $k_1, k_2 \in \mathbb{N}$ with $k_1 \leq k_2$, we have*

1. *N is $([k_1,k_2],R')$-sound for all $R \leq R' \leq R^+$ iff N is up-to $([k_1,k_2],[R,R^+])$-sound.*
2. *N is down-to $([k_1,k_2],[R^-,R])$-sound implies N is $([k_1,k_2],R')$-sound for all $R^- \leq R' \leq R$.*
3. *Let $R = R^+$. N is down-to $([k_1,k_2],[R^-,R])$-sound iff N is up-to, down-to $([k_1,k_2],[R^-,R^+])$-sound.*

Proof. (Sketch) It suffices to prove the three statements for k, $k_1 \leq k \leq k_2$.

(1) $\Rightarrow$: Let m_0 be the initial marking for (k,R')-soundness with $R' = R^+$ and m_0' be the initial marking for up-to $(k,[R,R^+])$-soundness. Clearly, we have $m_0|_{P_r} \geq m_0'|_{P_r}$. Let $m_0' \xrightarrow{\sigma} m'$. Then by the monotonicity of the firing rule and construction of E, we have $m_0 \xrightarrow{\sigma|_{T_N}} m$ and $m|_{P_r} \geq m|_{P_r}$. Thus, if we consider the projection of markings to N, then every marking that is reachable for up-to $(k,[R,R^+])$-soundness is also reachable for (k,R')-soundness.

$\Leftarrow$: Any resource vector R' within the interval can be reached by firing transitions $t_r^{\uparrow}$, for all $r \in P_r$. Then, every run in the composition for (k,R')-soundness can be replayed in the net for up-to $(k,[R,R^+])$-soundness.

(2) Similar argumentation as in the reverse implication of (1), but this time transitions $t_r^{\downarrow}$ and $\bar{t}_r^{\downarrow}$ have to be fired.

(3) $\Rightarrow$: Similar argumentation as for the implication of(1).

$\Leftarrow$: Use argumentation in (2) to decrease the resource vector to R. □

The next lemma gives a necessary condition for interval soundness, thereby justifying the restriction to well-defined compositions of N and E.

Lemma 14. (necessary condition). *Let N be an RCWF-net and $\langle E, m_E\rangle$ be an environment of N. If N is sound with $\langle E, m_E\rangle$, then $N \oplus E$ is well-defined.*

Proof. (1) k-soundness of the production net of N follows from [16, Cor. 4.1].

(2) Follows from [16, Thm. 4.4].

(3) Existence of an invariant I follows from [16, Thm. 4.8] and by the construction of E, we have $I + (c + d + e)$ is also an invariant.

(4) By [16, Thm. 4.8], a resource invariant exists for all $r \in P_r$. Moreover, we have the following invariant: $r + r^- + 2r^+ + 2r^{\uparrow} + 2\bar{r}_1^{\uparrow} + \bar{r}_2^{\uparrow} + r^{\downarrow} + \bar{r}_1^{\downarrow} + 2\bar{r}_2^{\downarrow}$, for $r \in P_r$. So the sum of these two invariants is the resource invariant we are looking for. □

Example 2. Consider the RCWF-net N_1 in Fig. 1(a) and its environment E (see Fig. 4(a) for the entire composition). The production net of N_1 is k-sound, for $k > 0$, and $i + p + q + f + c + d + e$ is a place invariant in the production net. Furthermore, no resource token is created in N_1, and $r + p + 2q + r^- + 2r^+ + 2r^{\uparrow} + 2\bar{r}_1^{\uparrow} + \bar{r}_2^{\uparrow} + r^{\downarrow} + \bar{r}_1^{\downarrow} + 2\bar{r}_2^{\downarrow}$ an invariant for resource place r. Thus, $N_1 \oplus E$ is well-defined. For the same reason, $N_2 \oplus E$ is well-defined. However, N_1 is not (k,R)-sound for all $k = R(r)$. For example, for $k = r = 2$, firing transition t twice yields a deadlock $[2 \cdot p]$. In contrast, N_2 is (k,R)-sound for $R(r) > 1$ and any k. This example exemplifies that well-definedness is only a necessary condition for interval soundness.

4 Deciding Interval Soundness

Definition 12 (interval soundness) gives an immediate decision procedure: An RCWF-net N is sound with $\langle E, m_E \rangle$ if the set $\{m \mid m(e) = m_E(c) + m_E(d)\}$ of markings is a home-space in $N \oplus E$ or respectively $m = [(m_E(c) + m_E(d)) \cdot e]$ is a home-marking in the projection of reachable markings of $N \oplus E$ to the places of N.

Theorem 15. (decision I) *Let N be an RCWF-net and $\langle E, m_E \rangle$ be an environment of N such that $N \oplus E$ is well-defined. Then, N is sound with $\langle E, m_E \rangle$ if the set $\{m \mid m(e) = m_E(c) + m_E(d)\}$ of markings is a home-space in $N \oplus E$.*

Note that we have one decision algorithm, for every instance of interval soundness. As checking a home-space property is decidable [12], we can conclude:

Theorem 16. (decidability) *Let N be an RCWF-net and $\langle E, m_E \rangle$ be an environment of N. Checking whether N is sound with $\langle E, m_E \rangle$ is decidable.*

In the literature, soundness is often reduced to showing that the short-circuited (RC)WF-net is live and bounded. The reduction works if the transition t_s consumes all k cases from the place f and produces k tokens on the place i. Figure 3 illustrates this. The drawback of this construction is that it requires to check a different net for every k. As we consider N with an environment, we propose the following generic reduction to liveness and boundedness:

Theorem 17. (decision II) *Let N be an RCWF-net and $\langle E, m_E \rangle$ be an environment of N such that $N \oplus E$ is well-defined. Let E_s be obtained from E by adding a transition t_s with $W((e, t_s)) = X(c) + X(d)$, $W((t_s, d)) = X(d)$, and $W((t_s, c)) = X(c)$. Then, N is sound with $\langle E, m_E \rangle$ iff all transitions $T_N \uplus \{t_s\}$ of $N \oplus E_s$ are live.*

Proof. $\Rightarrow$: As N does not have redundant places, it does not have dead transitions in its production net and t_s is not dead either, so we conclude that all transitions in $T_N \uplus \{t_s\}$ are live.

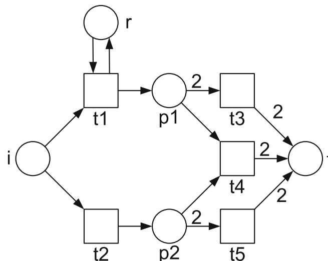


Fig. 3. The RCWF-net N is not $(3, 1)$-sound: The firing sequence $t_1 t_2 t_4 t_1$ yields the marking $[2 \cdot f, p_1]$ which is a deadlock. Nevertheless, the short-circuited net of N is bounded and live. However, if the transition t_s consumes all k cases from the place f and produces k token on the place i, then the short-circuited net of N is not live

$\Leftarrow$: From liveness of the transition t_s, we conclude that it is always possible to reach a marking m with $m(e) \geq m_E(c) + m_E(d)$. The number of tokens in e at m cannot be greater than $m_E(c)+m_E(d)$ by the invariant in Definition 10(3). Moreover, all places of the production net N_p of N are unmarked in m by the k-soundness of N_p. Firing t_s at m yields m' where the places c and d contain the same number of tokens as in the initial marking m_Z. Because of the invariants covering all places of E (Definition 10(3) and (4)), we conclude that m' is reachable from m_Z. Hence, markings m are a home-space in $N \oplus E_s$, and N is sound with $\langle E, m_E\rangle$. □

Example 3. Using Theorem 17, we can show that the RCWF-net N_1 is not (k, R)-sound whereas the RCWF-net N_2 is.

5 Repairing Interval Unsound RCWF-Nets

In the previous section, we presented an algorithm to decide interval soundness of an RCWF-net N. However, designing an interval-sound workflow or adjusting a workflow if some functionality or the environment has been changed is a nontrivial and error-prone task even for experienced process designers. In order to support process designers, we introduce an approach to *repair* an interval-unsound RCWF-net N if possible so that interval soundness is achieved by design. Clearly, the repaired workflow should be seen as a suggestion to the process designer rather than the ultimate solution.

Requiring the composition $N \oplus E$ to be well-defined reduces the cause of unsoundness to a deadlock or a livelock due to the lack of resources during the production process (see Lemma 11). To repair an RCWF-net N, we therefore propose to automatically construct a *controller* C that controls those transitions of N that produce tokens on or consume tokens from a resource place. This way, we control the order in which certain tasks may occur and prevent the workflow from getting stuck.

Technically, a controller is a labeled Petri net C and will be composed with $N \oplus E$ by merging transitions of N only. These merged transitions of N are then controlled by C in the composition. Another technicality that we leave out in the following is ensuring that the nodes of E and C are pairwise disjoint.

Definition 18. (controller, repairable) Let N be an RCWF-net and $\langle E, m_E\rangle$ be an environment of N. A labeled Petri net C is a *controller* of $N \oplus E$ if C and N are s-composable and replacing Z in Definition 12 with $C \| N \oplus E$ yields soundness of N with $\langle E, m_E\rangle$. If there exists a controller of $N \oplus E$, then $N \oplus E$ is *repairable.*

The following algorithm synthesizes a controller of $N \oplus E$. It takes the state space of N and the environment E as its input, and it outputs an LTS which can, in a next step, be easily transformed into a labeled Petri net.

Definition 19. (controller construction) Let N be an RCWF-net and and $\langle E, m_E\rangle$ be an environment of N such that $Z = N \oplus E$ is well-defined and has

a finite state space. Let $\Sigma \subseteq \Sigma_N$ be the set of synchronized actions, and let $TS_Z = \langle Q_Z, \delta_Z, \hat{q}_Z, \Sigma \rangle$ be the LTS of Z after relabeling all actions $x \in \Sigma_N \setminus \Sigma$ to τ. Define a sequence of LTSs TS^i, $i = 0, 1, \ldots$ inductively as follows:

Base : $TS^0 = \langle \mathcal{Q}^0, \delta^0, Q_0, \Sigma \rangle$ with
- $\mathcal{Q}^0 = 2^{Q_Z}$,
- $\delta^0 = \{(Q, x, Q') \in \mathcal{Q}^0 \times \Sigma \times \mathcal{Q}^0 \mid Q' = \{q'_Z \mid \exists q_Z \in Q : q_Z \xrightarrow{\tau^* x \tau^*} q'_Z\}\}$,
- $Q_0 = \{q_Z \mid \hat{q}_Z \xrightarrow{\tau^*} q_Z\}$.

Step : $TS^{i+1} = \langle \mathcal{Q}^{i+1}, \delta^{i+1}, Q_0, \Sigma \rangle$ with
- $\mathcal{Q}^{i+1} = \mathcal{Q}^i \setminus \{Q \in \mathcal{Q}^i \mid \exists (q_Z, Q) \in TS_Z \| TS^i : (q_Z, Q) \not\xrightarrow{*} (q'_Z, Q') \wedge q'_Z|_{\{e\}} = k_1 + k_2\}$[1],
- $\delta^{i+1} = \delta^i \cap (\mathcal{Q}^{i+1} \times \Sigma \times \mathcal{Q}^{i+1})$.

Let j be the smallest number with $TS^j = TS^{j+1}$. If $Q_0 \in \mathcal{Q}^j$, then the corresponding labeled Petri net C of TS^j is a controller of $N \oplus E$.

The construction of C allows some level of flexibility: We do not assume all transitions of N to be controllable as we restrict the set of labels of TS^j in the construction to a subset Σ of the alphabet Σ_N. That way, we take into account that not all tasks in a workflow can be controlled. We assume the nodes of C and E to be pairwise disjoint; that is, the controller cannot control actions of the environment. (If one would find it possible to control actions of the environment, we could adapt our construction by labeling certain transitions of E and adding those actions to the alphabet of C.)

Note that C is not necessarily a WF-net, because it may have more than one sink place; thus, the composition $N \| C$ is not an RCWF-net but a labeled Petri net.

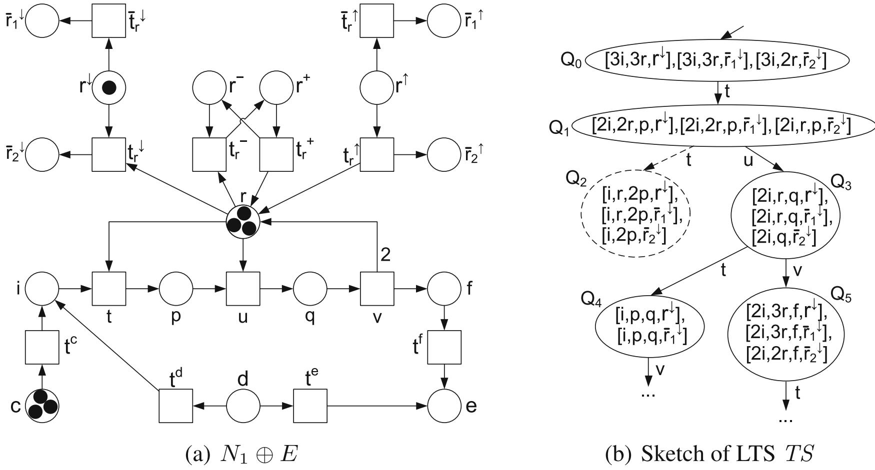
Our main result of this section is that a composition $N \oplus E$ is repairable if and only if the algorithm in Definition 19 outputs an LTS with at least one state.

Theorem 20. (justification) *Let N be an RCWF-net, $\langle E, m_E \rangle$ be an environment of N, and C be the labeled Petri net constructed according to Definition 19. Then, C exists iff $N \oplus E$ is repairable.*

Proof. $\Rightarrow$: Suppose C is a controller of $N \oplus E$. As TS^0 is the largest structure, there must be a largest i such that there exists a simulation relation of (the state space) of C by TS^i. If there is no simulation relation of C by TS^{i+1}, then $(N \oplus E) \| C$ must violate soundness because this is the reason for removing further states from TS^i.

$\Leftarrow$: The construction of TS^j and thus of C terminates because finiteness of TS_Z ensures that TS^0 is also finite and only removes states and transitions from TS^0 when constructing TS^j. If the resulting TS^j is nonempty, it must be a controller of $N \oplus E$ because all reasons not to be a controller have been erased: We removed all states from which N cannot reach a final state and we iteratively check this. □

[1] For the sake of readability, we see the state q'_Z as its corresponding marking in Z.



(a) $N_1 \oplus E$ (b) Sketch of LTS TS

Fig. 4. Illustration of the controller construction (see Definition 19) for $N_1 \oplus E$, assuming an initial marking $[3 \cdot i, 3 \cdot r, r^{\downarrow}]$ (i.e., the transition t^c has been fired three times) for purposes of simplification. The state Q_2 contains a deadlock and will be removed in one of the iterations

So, Definition 19 yields the *most permissive* controller of $N \oplus E$. Other, more specific controllers that have less behavior can also be constructed, for example, by assigning costs to each transition in N and constructing a controller that has the least cost.

Example 4. Figure 4 illustrates the controller construction for $N_1 \oplus E$. For the sake of readability, we keep the size of each controller state (i.e., the number of markings $N_1 \oplus E$ can be in) small by choosing $[3 \cdot i, 3 \cdot r, r^{\downarrow}]$ to be the initial marking of $N_1 \oplus E$—the state space of the controller remains the same. Initially, $N_1 \oplus E$ can be in any of the three markings of the initial state Q_0 of TS. Firing the transition t, yields the three markings depicted in the state Q_1 of TS. The state Q_2 (depicted by a dashed frame) is removed in one of the iterations of the construction, because the marking $[i, 2 \cdot p, \bar{r}_2^{\downarrow}]$ is a deadlock and no final marking. The complete LTS TS, the controller, has 12 states and realizes $tu(tv+vt)u(tv+vt)uv$. Composing the resulting labeled Petri net C of TS with $N_1 \oplus E$ yields a sound net $C \| N_1 \oplus C$. Note that the composition operator $\|$ requires one copy for each occurrence of the transition t, u, and v in C.

6 Related Work

The verification of soundness for RCWF-nets has been investigated by many researchers. On the one hand, interval soundness is a more restrictive instance compared to soundness in [21,13,5], as we assume the number of cases and resources to be fixed within an interval. On the other hand, it is more general

because we assume resources to be shared among workflows rather than internal to a workflow. We incorporated this in our model by enriching the model of RCWF-nets as proposed in [6,13] with a generic environment modeling the resource perspective of a WF-net. Moreover, we are neither restricted to one resource type as [13] nor to certain subclasses of WF-nets as [5]. Resource problems with an unbounded number of resource items have been studied in [9].

RCWF-nets can be seen as parameterized (or multi-threaded) systems with two parameters: the number of cases to be executed and the number of available resources. Verification of parameterized systems is a popular topic, but must approaches investigate safety properties with unbounded parameters (e.g., [18]) whereas we assume fixed bounds but consider with soundness a liveness property. A resource interface in [11] defines a safety property over the resources for open system; we defined the generic environment E and hence deal with a closed system. There also exist extensions of the temporal logics CTL and ATL to reason about resources [10,4]. Although the problem instances considered in this paper can be expressed in terms of those logics, verification would require to check the system for all parameters.

Our approach to repair unsound workflows is based on classical controller synthesis [20] and has been defined for Petri nets and soundness in [22]. For an overview of Petri net-based controller synthesis approaches, we refer to [19]. Most of these works focus on the net structure and properties different from soundness; moreover, the main application are manufacturing systems where one tries to construct a scheduler. In contrast, we construct a controller such that the net is robust and thus sound.

7 Conclusion

We investigated the correctness of workflows with shared resources, called interval soundness. To do so, we proposed to enrich the workflow model with a generic environment capturing the resource perspective. An instance of this generic environment models a specific environment that specifies an interval of workflow instances to be created and available resources for each resource type. The generic environment generalizes the existing workflow model extended with resources and captures environments of practical relevance. To decide interval soundness for every instance of the generic environment, we presented two decision procedures, both using invariant properties of the environment. Furthermore, we showed a way to support the design of correct workflows by automatically synthesizing a controller such that the composition of the workflow and the controller is interval sound.

In ongoing work, we are interested in determining a smallest resource vector (based on given requirements) that guarantees soundness. Likewise, we aim at determining the largest resource vector that guarantees soundness or proving that increasing some resource vector does not influence the soundness result. Constructing more specific controllers by assigning costs to transitions is another direction of future work.

References

1. van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
2. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
3. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge (2002)
4. Alechina, N., Logan, B., Nga, N.H., Rakib, A.: Resource-bounded alternating-time temporal logic. In: AAMAS 2010, pp. 481–488. IFAAMAS (2010)
5. Barkaoui, K., Benayed, R., Sbai, Z.: Workflow Soundness Verification Based on Structure Theory of Petri Nets. International Journal of Computing & Information Sciences 5(1), 51–62 (2007)
6. Barkaoui, K., Petrucci, L.: Structural Analysis of Workflow Nets with Shared Ressources. In: WFM 1998, pp. 82–95 (1998)
7. Best, E., Darondeau, P.: Separability in persistent petri nets. Fundam. Inform. 113(3–4), 179–203 (2011)
8. Best, E., Esparza, J., Wimmel, H., Wolf, K.: Separability in conflict-free petri nets. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 1–18. Springer, Heidelberg (2007)
9. Brázdil, T., Jančar, P., Kučera, A.: Reachability games on extended vector addition systems with states. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010)
10. Bulling, N., Farwer, B.: Expressing properties of resource-bounded systems: The logics rTL* and RTL. In: Dix, J., Fisher, M., Novák, P. (eds.) CLIMA X. LNCS, vol. 6214, pp. 22–45. Springer, Heidelberg (2010)
11. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
12. Escrig, D.F., Johnen, C.: Decidability of home space property. LRI report 503, Université Paris-Sud (1989)
13. van Hee, K.M., Serebrenik, A., Sidorova, N., Voorhoeve, M.: Soundness of resource-constrained workflow nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 250–267. Springer, Heidelberg (2005)
14. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Soundness and separability of workflow nets in the stepwise refinement approach. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 337–356. Springer, Heidelberg (2003)
15. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Generalised soundness of workflow nets is decidable. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 197–215. Springer, Heidelberg (2004)
16. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Resource-constrained workflow nets. Fundam. Inform. 71(2–3), 243–257 (2006)
17. Juhás, G., Kazlov, I., Juhásová, A.: Instance deadlock: A mystery behind frozen programs. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 1–17. Springer, Heidelberg (2010)
18. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010)

19. Li, Z.W., Wu, N.Q., Zhou, M.C.: Deadlock control of automated manufacturing systems based on petri nets–a literature review. IEEE Transactions on Systems, Man, and Cybernetics, Part C 42(4), 437–462 (2012)
20. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control and Optimization 25(1), 206–230 (1987)
21. Sidorova, N., Stahl, C.: Soundness for resource-constrained workflow nets is decidable. IEEE Transactions on Systems, Man and Cybernetics, Part A 43(3), 724–729 (2013)
22. Wolf, K.: Does my service have partners? In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 152–171. Springer, Heidelberg (2009)

Statistical Model Checking of a Clock Synchronization Protocol for Sensor Networks*

Luca Battisti, Damiano Macedonio, and Massimo Merro

Dipartimento di Informatica, Università degli Studi di Verona, Verona, Italy

Abstract. This paper uses the statistical model checking tool in the UPPAAL toolset to test the robustness of a distributed clock synchronization algorithm for wireless sensor networks (WSN), in the case of lossy communication, i.e., when the WSN is deployed in an environment with significant multi-path propagation, leading to interference. More precisely, the robustness of the gMAC protocol included in the Chess WSN platform is tested on two important classes of regular network topologies: cliques (networks with full connectivity) and small grids (where all nodes have the same degree). The paper extends previous work by Hedaraian et al. that only analyzed this algorithm in the ideal case of non-lossy communication, and only in the case of cliques and line topologies. The main contribution is to show that the original clock synchronization algorithm is not robust to changing the quality of communication between sensors. More precisely, with high probability the algorithm fails to synchronize the nodes when considering lossy communication over cliques of arbitrary size, as well as over small grid topologies.

1 Introduction

Wireless sensor networks (WSNs) are (possibly large-scale) networks of sensor nodes deployed in strategic areas to gather data. Sensor nodes collaborate using wireless communications with an asymmetric many-to-one data transfer model. Typically, they send their sensed data to a sink node which collects the relevant information. WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature); they are used as embedded systems (e.g., biomedical sensor engineering, smart homes) or mobile applications (e.g., when attached to robots, soldiers, or vehicles).

In wireless sensor networks, the basic operation is *data fusion*, whereby data from each sensor is agglomerated to form a single meaningful result. The fusion of individual sensor readings is possible only by exchanging messages that are timestamped by each sensor's local clock. This mandates the need for a common notion of time among the sensors which is achieved by means of so called *clock synchronization protocols* [13,15].

* Work partially supported by the PRIN 2010-2011 project "Security Horizons".

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 168–182, 2013.
DOI: 10.1007/978-3-642-40213-5_11,
© IFIP International Federation for Information Processing 2013

In this paper we do model checking of a distributed algorithm of clock synchronization for WSNs that has been developed by the Dutch company CHESS [12]. In order to realize an energy efficient communication mechanism, CHESS developed a gossip-based MAC algorithm [14] (abbreviated gMAC) which is responsible for regulating the access to the wireless shared channel. Here we are interested in verifying the robustness of the gMAC algorithm in the presence of *packet loss*. Packet loss is particularly relevant in wireless sensor networks which are deployed in environments with significant multi-path distortion (when part of the signal goes to the destination while another part bounces off an obstruction and then goes on to the destination). Most of sensor platforms do not have enough frequency diversity to reject multi-path propagation.

Our work has been strongly inspired by a recent analysis [8] of the gMAC synchronization protocol on clique and line topologies, in the ideal case of non-lossy communication. In the case of line topologies, paper [8] shows that the protocol fails to synchronize all nodes, when the number of nodes grows. On the other hand, on clique topologies the protocol behaves quite well and the paper provides constraints on *guard times* (delays added before and after the transmission of sync messages) to guarantee clock synchronization, independently on the clique size. In [8] the protocol is modeled as a network of timed automata and verified using the UPPAAL model checker [2,3]. However, the model in [8] does not incorporate several features such as dynamic slot allocation, uncertain communication delays, and unreliable radio communication. In the current paper we extend their analysis by adopting a *probabilistic model* of radio communication that takes into account message loss according to the measurement of packet delivery suggested in [17]. As our model is a network of probabilistic timed automata, we decided to do our analysis by applying *Statistical Model Checking* (SMC) [7] within the UPPAAL toolset [2,3]. SMC consists in monitoring a proper number of runs of the system and then applying a statistical algorithm to obtain an estimate of the result of the desired query.

Our analysis shows that low guard times (within the safety range proposed in [8]) are not sufficient to guarantee clock synchronization in clique topologies of arbitrary size. More precisely, in the case of lossy communication, the size of the clique *does* play a crucial role in the effectiveness of the protocol: the bigger is the clique the higher must be the guard time to ensure clock synchronization with high probability. Here it is important to notice that guard times cannot be arbitrary increased without dramatically affecting the duration of the battery life of the sensor nodes [1]. Finally, we move our analysis on grid topologies, with increasing neighbor degree, to better simulate a uniform node distribution of sensor nodes in a given area. Our simulations show that high values of the guard times may be not sufficient to guarantee clock synchronization in the presence of message loss, even in small 5×5 grid networks. On the other hand, we observe that the efficiency of the protocol improves when the number of neighbours, and hence node connectivity, increases.

Outline Section 2 introduces the gMAC protocol. Section 3 illustrates the corresponding UPPAAL probabilistic model. Section 4 details our analysis on cliques

and grid topologies. Section 5 concludes the paper with final remarks, future and related work.

2 The gMAC Protocol

The gMAC protocol is a Time Division Multiple Access (TDMA) protocol, where time is divided into fixed length *frames*, and each frame is subdivided into *slots*. Slots can be either *active* or *idle*. During active slots, a node is either listening for incoming messages from neighbouring nodes (RX slot) or it is sending a message (TX slot). During idle slots a node is switched to energy saving mode. Active slots are gathered in a contiguous sequence placed at the beginning of each frame.

Structure of a time frame:

RX	···	RX	TX	RX	···	RX	idle slots

Structure of an active slot:

← g →	sending/receiving	← t →

Since energy efficiency is a major concern in the design of wireless sensor networks, the number of active slots is typically much smaller than the total number of slots. In the implementation of gMAC the number of slots within a frame is 1129 out of which 10 are active. A node can only transmit a message once per time frame in its TX slot. If two neighbouring nodes choose the same send slot then a communication collision will occur in the intersection of their transmission ranges preventing message delivery. In the original protocol a node randomly chooses an active slots as *send slot* (TX slot) considering all other active slots as *receive slots* (RX slots). However, for the sake of simplicity, as in [8], in our analysis we assume that the TX slots are fixed and have been chosen in such a way that no collision occurs.

In order to ensure that when a node is sending all its neighbours are listening, *guard times* are introduced. This means that each sender waits for some time (g clock cycles) at the beginning of its TX slot to ensure that all its neighbours are ready to receive messages; similarly, a sender does not transmit for a certain amount of time (t clock cycles) at the end of its TX slot. Guard times cannot be arbitrary increased without dramatically affecting the duration of the battery life of the sensor nodes [1]. So, the choice of proper guard time values is crucial in the protocol design. In the current implementation, each slot consists of 29 clock cycles, out of which 18 cycles are used as guard time.

The CHESS sensor nodes come equipped with a 32 kHz crystal oscillator that drives an internal clock used to determine the beginning and the end of each slot. Sensor nodes are also equipped with an ATMega64 micro-controller and a Nordic nRF24L01 [10] packet radio. Depending on the environment, the Nordic nRF24L01 radio has a transmission range between $0.5m$ and $50m$, For the sake of simplicity we assume that all nodes have the same transmission range; this means that the transmission between nodes is assumed to be symmetric.

3 UPPAAL Probabilistic Model for gMAC

In this section, we provide a small extension of the UPPAAL model for gMAC of [8] in which a probabilistic choice to model message loss is introduced. The model assumes a finite, fixed set of sensor nodes $\mathsf{Nodes} = \{0, \ldots, \mathsf{N} - 1\}$. The behaviour of each individual node $i \in \mathsf{Nodes}$ is described by means of three different timed automata: **Clock**(i), **WSN**(i), **Synchronizer**(i). Automaton **Clock**(i) models the hardware clock of the node, **WSN**(i) takes care of sending messages, and **Synchronizer**(i) re-synchronizes the hardware clock upon receipt of a message. The automaton **Synchronizer**(i) is the only one where probabilities are introduced to model packet loss. The complete model consists of the composition of the three automata **Clock**(i), **WSN**(i) and **Synchronizer**(i), for each $i \in \mathsf{Nodes}$.

For each node i there are two state variables: $\mathsf{clk}[i]$, which records the value of the hardware clock (initially 0), and $\mathsf{csn}[i]$, which records the current slot number (initially 0). Furthermore, there are two broadcast channels: $\mathsf{tick}[i]$, used to synchronize the activities within the node i, and $\mathsf{start_message}[i]$, used to inform all neighbours of the beginning of i's transmission. Table 1 reports the protocol parameters.

Figure 1 depicts the automaton **Clock**(i) of [8] modeling the hardware clock of node i. The local clock variable x measures the time between two consecutive clock ticks. A $\mathsf{tick}[i]!$ action is enabled when x reaches the value min and must fire before x reaches the value max. When the action $\mathsf{tick}[i]!$ occurs the variable x is reset to 0 and the variable $\mathsf{clk}[i]$ is incremented by 1. As explained in [8], the state variable $\mathsf{clk}[i]$ is reset after k_0 clock ticks for the model checking to become feasible. A realistic clock drift rate is about 20 ppm (parts-per-million). Such a rate is achieved in the model by setting $\mathsf{min} = 10^5 - 2$ and $\mathsf{max} = 10^5 + 2$. In the model of [8] ticks may nondeterministically occur within the time interval $[\mathsf{min}, \mathsf{max}]$; thus the delay between two $\mathsf{tick}[i]!$ actions is nondeterministic. The stochastic semantics for timed automata of UPPAAL SMC excludes nondeter-

Table 1. Protocol parameters

Parameter	Description	Constraints
N	number of nodes	$0 < \mathsf{N}$
C	number of slots in a time frame	$0 < \mathsf{C}$
n	number of active slots in a time frame	$0 < \mathsf{n} \leq \mathsf{C}$
k_0	number of clock ticks in a time slot	$0 < \mathsf{k}_0$
$\mathsf{csn}[i]$	current slot number for node i	$0 \leq \mathsf{csn}[i] < \mathsf{n}$
$\mathsf{tsn}[i]$	TX slot number for node i	$0 \leq \mathsf{tsn}[i] < \mathsf{n}$
$\mathsf{clk}[i]$	clock value for node i	$0 \leq \mathsf{clk}[i] < \mathsf{k}_0$
g	guard time	$0 < \mathsf{g}$
t	tail time	$0 < \mathsf{t}$
min	minimal time between two clock ticks	$0 < \mathsf{min}$
max	maximal time between two clock ticks	$\mathsf{min} \leq \mathsf{max}$
loss	*message loss probability*	$0 \leq \mathsf{loss} \leq 100$

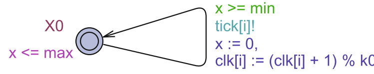


Fig. 1. Clock(i)

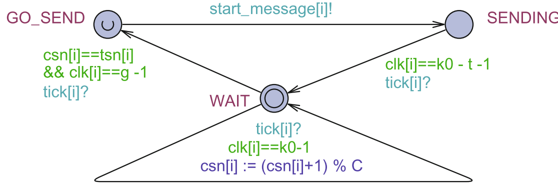


Fig. 2. WSN(i)

minism; thus the delay between two tick[i]! actions is implemented as a uniformly distributed stochastic delay. This may be non-realistic in general: clocks usually run either too fast or too slow for long periods of time, due to environmental differences. However in our analysis we will focus on small networks and we will assume all sensors being in the same environmental conditions. Thus we exploit UPPAAL SMC's random clock speed jitter.

Figure 2 describes automaton **WSN**(i) of [8] devoted to message sending. The automaton waits in the initial location WAIT until the current slot number csn[i] equals the TX slot tsn[i], and g ticks occur in that slot. Then the automaton moves to the location GO_SEND which is immediately left by performing a start_message[i]! action. This action leads the automaton to the location SENDING. The automaton remains in this location until the beginning of the tail interval (which starts after $\mathsf{k}_0 - \mathsf{t}$ ticks). Then the automaton returns to the location WAIT where it increments the current slot number csn[i] every k_0 ticks.

Figure 3 contains the automaton **Synchronizer**(i) which is devoted to synchronize the hardware clock. We enrich the corresponding automaton of [8] with a simulation of message loss on the channel start_message[i]. The UPPAAL model checker features branching edges with associated weights for the probabilistic extension. Thus we define an integer constant loss, with $0 \leq \mathsf{loss} \leq 100$, and every node can either lose a message with weight loss or receive it with weight (100$-$loss). The automaton **Synchronizer**(i) waits in its initial location S0 until it detects, in an active slot (csn[i] $<$ n), the beginning of a new message from a neighbor j (action start_message[j]?). When this happens the automaton moves to a committing location C and it immediately goes to a branching edge where: (i) with weight loss it returns in its initial location S0 and (ii) with weight 100$-$loss it goes to location S1. Case (i) formalizes the loss of the starting message from node j, while case (ii) formalizes the reception of the same message. Notice that UPPAAL requires input determinism to ensure that the system to be tested always produces the same outputs on any given sequence of inputs. Thus we need an extra intermediate location instead of branching immediately on

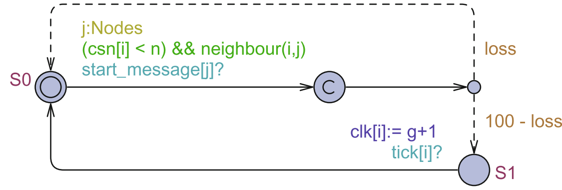


Fig. 3. Synchronizer(i)

the $\mathsf{start_message}[j]!$ action. Notice also that the $\mathsf{start_message}[j]!$ action occurs exactly when $\mathsf{clk}[j] = \mathsf{g}$ (as a result of the synchronization of automata **WSN**(j) and **CLOCK**(j)); while the node i, by means of automaton **Synchronizer**(i), resets the variable $\mathsf{clk}[i]$ to $\mathsf{g} + 1$ after a further tick. The guard $\mathsf{neighbor}(i, j)$ indicates that i and j are in the transmission range of each other. Formally, $\mathsf{neighbor}()$ is a symmetric function related to the slot allocation in the following manner [8]:

$$\begin{aligned} \mathsf{neighbor}(i,j) &\Longrightarrow \mathsf{tsn}[i] \neq \mathsf{tsn}[j] \\ \mathsf{neighbor}(i,j) \wedge \mathsf{neighbor}(i,k) &\Longrightarrow \mathsf{tsn}[j] \neq \mathsf{tsn}[k] \end{aligned} \tag{1}$$

This means that whenever two nodes are neighbours or have a common neighbor then they must have distinct TX slot numbers. The function $\mathsf{neighbor}()$ is helpful to provide a formal definition of synchronized sensor networks. Intuitively, a sensor network is said to be synchronized if, whenever a node is sending in a given slot number then all neighbouring nodes are in the same slot number.

Definition 1. *A network is said to be synchronized if for all reachable states* $(\,\forall i, j \in \mathsf{Nodes}\,)(\,(\,\mathsf{SENDING}_i \wedge \mathsf{neighbor}(i,j)\,) \Longrightarrow (\,\mathsf{csn}[i] = \mathsf{csn}[j]\,)\,)$.

4 Our Analysis

UPPAAL Statistical Model Checker [7] evaluates properties on execution runs of a network of probabilistic timed automata. The execution time of these runs is represented by a variable `time`. It is left to the user to set a bound to this variable. In particular, fixed a constant value `bound`, the Statistical Model Checker can reply to queries of the following shape

```
Pr[time<=bound] (<> expr)
```

by performing an adequate number of runs to estimate the probability to reach a state which satisfies the property `expr`, within the time `bound`. The user must fix two main statistical parameters, α and ε, both in the real interval $]0, 1[$. The answer provided by the tool is a confidence interval $[p - \varepsilon, p + \varepsilon]$ where α represents the probability of the answer of being wrong. The higher is the precision of the analysis the bigger must be the number of runs performed by the simulator. Thus the waiting time for a reply of a query depends both on

the length of runs, i.e. on the parameter bound, and on the statistical parameters ε and α.

In order to make feasible our analysis we try to understand if we can change some system parameters without affecting the quality of the analysis. In particular, we focus on the parameter C that is the number of slots composing a frame. The effectiveness of any synchronization protocol is crucially based on the exchange of some timing information to synchronize neighbor nodes. Said in other words, the longer nodes remain silent the quicker they get out of sync, because they do not get enough information to synchronize with each other. As consequence, once fixed the number of active slots, if the system gets out of sync with probability p, for a certain value of C, then the same system will get out of sync more quickly (or with higher probability) for a bigger value of C.

The parameter loss expresses the probability of message loss at the physical level due to the unreliability of the wireless medium. In our analysis, we will instantiate loss according to the results appeared in [17], where packet delivery performances of WSNs have been studied at physical layer under different transmission powers and physical-layer encodings. In that analysis, 60 Mica motes have been used to measure packet delivery under three different environmental settings: office building, open parking lot, habitat with moderate foliage. Under these settings, results show that the physical layer contributes to the packet-delivery performance, which is defined as the fraction of packets not successfully received by the receiver within a time window.

For the sake of simplicity, all nodes of our networks will be instantiated with the same value of the parameter loss. According to [17] this parameter will be set to: 10, to approximate the average message loss in a parking lot; 20, to model the average message loss in a office building; and 30, which represents the average message loss in an habitat setting with moderate foliage.

4.1 Verifying Clique Topologies

Paper [8] derives necessary and sufficient constraints on the guard times to guarantee the correctness of the protocol on clique topologies in the case of perfect communication. These constraints depend on the clock ratio $\mathsf{min}/\mathsf{max}$, on the parameter k_0 and on the maximal distance M between two transmitting slots[1]; they do not depend on the size N of the network. They are:

$$\begin{aligned} \mathsf{g} &> \left(1 - \tfrac{\mathsf{min}}{\mathsf{max}}\right) \cdot \mathsf{M} \cdot \mathsf{k}_0 + \tfrac{\mathsf{min}}{\mathsf{max}} \\ \mathsf{g} &< \left(1 - \tfrac{\mathsf{max}}{\mathsf{min}}\right) \cdot \mathsf{M} \cdot \mathsf{k}_0 + \mathsf{k}_0 - 2 \\ \mathsf{t} &> \left(1 - \tfrac{\mathsf{min}}{\mathsf{max}}\right) \cdot (\mathsf{k}_0 - \mathsf{g}) + \tfrac{\mathsf{min}}{\mathsf{max}} \end{aligned} \tag{2}$$

From an analysis of these conditions, paper [8] demonstrates that guard time values $\mathsf{g} = \mathsf{t} = 3$ are sufficient to guarantee clock synchronization in a clique of arbitrary size.

[1] For a formal definition of parameter M we refer to [8].

Table 2. On the parameter C

g	N	C	bound	frames	p	ε	α
3	10	12	$0.07 \cdot 10^9$	2	0.025	0.02	0.01
3	10	48	$0.28 \cdot 10^9$	2	0.029	0.02	0.01
3	10	192	$1.12 \cdot 10^9$	2	0.031	0.02	0.01
3	10	336	$2.00 \cdot 10^9$	2	0.031	0.02	0.01
4	15	17	$0.50 \cdot 10^9$	10	0.022	0.01	0.05
4	15	34	$1.00 \cdot 10^9$	10	0.027	0.01	0.05
4	15	68	$2.00 \cdot 10^9$	10	0.027	0.01	0.05

Here we want to demonstrate that, in the presence of packet loss, the size of the clique network *does* play a crucial role. In particular, the bigger is the clique the higher must be the value of g (and t) to ensure clock synchronization. Said in other words: *in the presence of message loss, fixed a value of* g *(and* t*), there is always a clique which gets out of sync with high probability.*

For networks with full connectivity clock synchronization means that all nodes of the network agree on the current slot. As a consequence, Definition 1 can be rephrased as in [8] in the following manner:

Definition 2. *A clique network is said to be synchronized if for all reachable states it holds the following:* $(\forall i, j \in \mathsf{Nodes})(\mathsf{SENDING}_i \Longrightarrow \mathsf{csn}[i]{=}\mathsf{csn}[j])$.

So, in order to estimate the probability of going out of synchronization we will use UPPAAL SMC to perform the following quantitative check:

```
Pr[time<=bound] (<> exists(i:Nodes) exists(j:Nodes)
                    (WSN(i).SENDING and not(csn[i]==csn[j]))        (3)
```

Simulation setting In our simulations on cliques, all protocol parameters will satisfy the constraints in (2). As in [8], the guard time t is chosen to be the same as g. Parameter $\mathsf{tsn}[i]$ is chosen equal to i, as fully connectivity implies a different TX slot for each node. We set $\mathsf{k}_0 = 29$. Unfortunately, we cannot set $\mathsf{C} = 1129$, as in the real implementation, because the length of the runs that can be analyzed by UPPAAL is limited: in order to avoid integer overflow, the parameter bound cannot overtake the value $2 \cdot 10^9$. This means that if we would keep C close to the real value, then our execution runs would last for just a single time frame and they would be too short to provide any significant result. According to the discussion done in the preface of this section we perform our analysis for low values of the parameter C. This modification does not affect our analysis. As an example, in Table 2 we consider cliques with $\mathsf{N} = 10, 15$, $\mathsf{loss} = 20$ and $\mathsf{g} = 3, 4$. We then perform the quantitative check (3) by varying C and keeping constant the number of observed time frames. The value p represents the center of the confidence interval computed by UPPAAL SMC. Every check required 6623 runs of the protocol and lasted for about four days on a Intel core i5-2420M CPU 2.30GHz with 6G RAM. We gradually increased the precision of the parameter

Table 3. Cliques and node number. Maximal run length. $\alpha = 0.05$, $\varepsilon = 0.025$

N = 10 frames: 60			N = 15 frames: 40			N = 20 frames: 30			N = 30 frames: 20		
g	loss	p	g	loss	p	g	loss	p	g	loss	p
3	10	0.059	3	10	0.100	3	10	0.133	3	10	0.236
3	20	0.386	3	20	0.560	3	20	0.692	3	20	0.851
3	30	0.787	3	30	0.931	3	30	0.972	3	30	0.992
4	10	0.000	4	10	0.000	4	10	0.003	4	10	0.005
4	20	0.025	4	20	0.046	4	20	0.063	4	20	0.106
4	30	0.155	4	30	0.243	4	30	0.325	4	30	0.464

ε in order to achieve an interval which does not include the value 0 as a reply; in other words we have looked for lower bounds $p - \varepsilon > 0$.

Table 2 outlines that *when the number of time frames is fixed then the probability of going out of sync for the system does not decrease when the parameter* C *increases* (similar results can be obtained for different values of N, g, loss and for different topologies). As a consequence, our simulations provide a lower bound of the probability of getting out of sync in a setting with C = 1129.

In Table 3 we study the behaviour of the protocol on cliques up to 30 nodes. We vary the number of nodes N, the guard time g and the parameter loss. Since we consider fully connected networks and the transmitting slots are grouped at the beginning of each time frame, we fix C = N+2, as in [8], to allow at least two idle slots at the end of each frame. We set the statistical parameters $\varepsilon = 0.025$, $\alpha = 0.05$ to have meaningful results. We check property (3) on the maximal run UPPAAL SMC can handle without incurring in integer overflow. The result of the quantitative check is represented by the probability p, which is the center of the confidence interval computed by UPPAAL SMC. Every check required 2952 runs of the protocol. In the following table we report the time required by our simulations on a Intel core i3-2310M CPU 2.10GHz with 4G RAM.

nodes	*time*
10	11 *hours*
15	1 *day* 7 *hours*
20	2 *days* 23 *hours*
30	9 *days* 4 *hours*

All runs in Table 3 are quite short (from 30 to 60 frames, depending on N); however, they are long enough to deduce some significant observation. For instance, we notice that once fixed the value of the guard time g, the probability of going out of sync increases when either N or loss increase. Moreover, once fixed both N and loss, the probability p decreases when the guard time g increases. Since the probability of going out of sync cannot decrease when going to longer runs, in Table 3 we compare probabilities associated to runs of different lengths. In particular, we notice that if we fix loss and g then the probability of getting out of sync increases when N increases. At the end of this section we will compare runs of the same length.

Table 4. Module comparison – $\mathsf{g} = 4$, $\alpha = 0.05$, run length 30 frames –

$\mathsf{N} = 5$			$\mathsf{N} = 10$		
$\mathcal{I}$	p	ε	$\mathcal{I}$	p	ε
$[0]$	0.019	0.01	$[0]$	0.015	0.01
$[0, \ldots, 4]$	0.113	0.030	$[0, \ldots, 4]$	0.113	0.030
$[0, \ldots, 9]$	0.665	0.050	$[0, \ldots, 9]$	0.677	0.030
$[0, \ldots, 14]$	0.827	0.050	$[0, \ldots, 14]$	0.816	0.030

The analysis provided in Table 3 says also that the protocol is certainly not suitable in certain scenarios. For instance, in a clique of at least 10 nodes with $\mathsf{g} = 3$ the system will get immediately out of sync with high probability if the loss probability is greater than 0.2. In other settings the results are not that strong. This is the case of a clique with 10 nodes, $\mathsf{g} = 4$ and $\mathsf{loss} = 20$. In this case, our analysis says that this system will get out of sync with probability 0.025. Such a value is ten times smaller than the loss probability, too small to conclude anything, at least in a so short run. Unfortunately, a priori, we cannot predict the behaviour of the system for longer runs as the probability p may increase or stabilize. In the following we will try to overcome this limitation.

UPPAAL SMC can simulate the behaviour of our systems on runs limited in size, called *execution modules*. At the beginning of an execution module all nodes are in the same time slot and with the same value in their clock variables. At the end of an execution module, UPPAAL SMC computes an estimate of the probability p to reach a state which does not satisfy Definition 2. This definition does not identify a single state of the system: nodes may have different clock values while still being in the same time slot. We claim that *the initial state, where all nodes begin the execution module with the same clock value, is the state which has the smallest probability to lead the system out of sync*. In order to support our argument, we provide an example in Table 4. We consider cliques of 5 and 10 nodes, with $\mathsf{g} = 4$, $\mathsf{C} = 7$ and $\mathsf{loss} = 20$. Table 4 shows experiments in which the system starts from a state that satisfies Definition 2 while internal clocks may have different values. The starting value of every internal clock is randomly chosen from a fixed interval $\mathcal{I}$ of clock values. Runs are 30 time frames long. We set $\alpha = 0.05$. It can be noticed that the smallest desync probability is obtained when the execution module starts in the initial state where all nodes have the same clock value. Similar results can be obtained for other values of N, g, C and loss.

In virtue of this observation, we can divide a long run in consecutive execution modules, all starting in the initial state. Then, we can derive by composition a lower bound of the probability of desynchronization for that run. Thus, if $[p - \varepsilon, p + \varepsilon]$ is the confidence interval provided by UPPAAL SMC after performing the quantitative check (3) within an execution module, then the probability of going out of sync within n execution modules is *at least*

$$1 - (1 - (p - \varepsilon))^n. \tag{4}$$

Table 5. Quantitative check on cliques with loss $= 20$ and $\alpha = 0.05$

g	N	C	$p - \varepsilon$	300 frames	600 frames	900 frames
3	10	12	0.361	≥ 0.893	≥ 0.989	≥ 0.999
3	15	17	0.535	≥ 0.998	≥ 0.999	≥ 0.999
3	20	22	0.667	≥ 0.999	≥ 0.999	≥ 0.999
3	30	32	0.826	≥ 0.999	≥ 0.999	≥ 0.999
4	15	17	0.021	≥ 0.156	≥ 0.273	≥ 0.373
4	20	22	0.038	≥ 0.321	≥ 0.539	≥ 0.687
4	30	32	0.081	≥ 0.718	≥ 0.920	≥ 0.980

Table 5 extends the results of Table 3 to longer runs which lasts for 300, 600 and 900 time frames, respectively. The forth column of Table 5 reports the lower bound of the confidence interval of an execution module. When $\mathsf{N} = 10, 15, 20, 30$ the execution module studied in Table 3 lasts for approximately 60, 40, 30 and 20 time frames respectively. Thus, by applying the formula (4) with $n = 5, 8, 10, 15$ we obtain a lower bound for the probability of being out of sync within 300 time frames in the cases of cliques with 10, 15, 20 and 30 nodes, respectively. Analogously when $n = 10, 15, 20, 30$ and $n = 15, 22, 30, 45$ we obtain a lower bound for the probability of being out of sync within 600 and 900 time frames, respectively.

As discussed at the beginning of this section, the values of Table 5 represent also lower bounds of the probability of desynchronization for the real implementation. In the real setting, with $\mathsf{C} = 1129$ and clock frequency of 32 kHz, a time frame lasts for about $1sec$. As a consequence, Table 5 expresses a lower bound of the probability of getting out of sync within 5, 10 and 15 *min*. Thus, when $\mathsf{g} = 3$ the probability of getting out of sync is high also for small networks enough for small networks (around 10 nodes), but when $\mathsf{N} = 15$ we have a probability of being out of sync of almost 0.4 in $15min$. When $\mathsf{N} = 20$ the probability reaches 0.7 in less than $15min$. When $\mathsf{N} = 30$ the probability reaches 0.7 in less than $5min$. These results outline an increasing of the desync probability when the number of nodes increases.

4.2 Verifying Grid Topologies

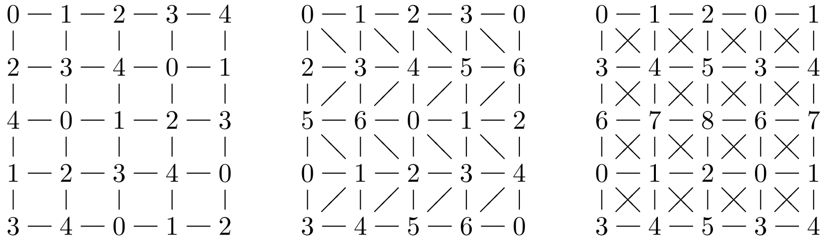
Clock synchronization in clique topologies has been studied in [8] as a first step towards more realistic topologies. Usually sensor nodes have a limited number of neighbours and do not have direct communication with the whole network. In this section, we study how the gMAC synchronization protocol behaves on regular topologies which better simulate a uniform node distribution in a given area[2]. In particular, we will focus on grid topologies where nodes have a uniform number of neighbours. Unlike cliques, there are no theoretical results suggesting how to choose protocol parameters to guarantee the synchronization of grid

[2] Regular topologies have been applied to WSNs to study coverage, connectivity and energy-efficiency.

networks in the case of non-lossy communication. The implementation of gMAC adopts an high guard time, $\mathsf{g} = 9$, to ensure synchronization in networks with arbitrary topologies. In this section, we study whether high values of g guarantee node synchronization in grid-based networks, in the case of lossy communication.

In our simulations we focus on a small sensor network where nodes are placed in a 5×5 grid, thus $\mathsf{N} = 25$. Unlike cliques, grid topologies do not need a different TX slot for each node: we can allocate the same TX slot to different nodes provided that when two nodes are neighbours or have a common neighbor then they get distinct TX slot numbers. According to the implementation of gMAC, where the number of TX slots is limited, we consider the minimum number of TX slots to be allocated to satisfy conditions (1). The number of transmission slots depends on the number of neighbours for each single node; if a node v has k neighbours then we need a TX slot in which v transmits and all its neighbours listen, and k distinct slots in which each neighbor transmits and v listens. Thus if k represents the maximum node degree, then we need at least $k+1$ TX slots.

In the following we analyze the behaviour of the protocol on grid topologies by considering three possible maximum node degrees: 4, 6 and 8. These grid networks require at least 5, 7 and 9 TX slots, respectively. Below we report the three topologies we consider along with a simple slot allocation which satisfies conditions (1) by using exactly $k+1$ TX slots, where k is the maximum node degree. The grid structures outlines the network topology while the identifiers $0, 1, \ldots$ show the TX slot allocated for the corresponding node.



As for cliques, our analysis does not loose in generality if we consider small values of C. Thus we pick $\mathsf{C} = 7, 9, 11$ for the three different cases, respectively. Depending on the maximum node degree, a single time frame is composed by $5, 7, 9$ TX slots plus 2 idle slots. TX slots are allocated according to the distributions depicted above. We set $\mathsf{k}_0 = 29$ and we vary the parameter loss, as done for cliques. Then, we apply UPPAAL SMC to perform the following quantitative check, according to Definition 1:

```
Pr[time<=bound](<> exists(i:Nodes) exists(j:Nodes)
  (neighbor(i,j) and WSN(i).SENDING and not(csn[i]==csn[j]))    (5)
```

Again, we consider the longest run UPPAAL SMC can handle to avoid integer overflow by setting $\mathtt{bound} = 2 \cdot 10^9$. This means that an execution module lasts for almost 100, 80 and 65 time frames when the maximum node degree is 4, 6 and 8, respectively. These are quite short runs, but long enough to conclude that

Table 6. Grids 5×5 and node degree. Maximal run length. $\alpha = 0.05$, $\varepsilon = 0.03$

max degree 4				max degree 6				max degree 8			
g	C	loss	p	g	C	loss	p	g	C	loss	p
6	7	0	0	6	9	0	0	6	11	0	0
6	7	10	0.03	6	9	10	0.01	6	11	10	0
6	7	20	0.11	6	9	20	0.07	6	11	20	0.05
6	7	30	0.45	6	9	30	0.25	6	11	30	0.19
7	7	0	0	7	9	0	0	7	11	0	0
7	7	10	0.01	7	9	10	0	7	11	10	0
7	7	20	0.06	7	9	20	0.03	7	11	20	0.02
7	7	30	0.28	7	9	30	0.18	7	11	30	0.11

Table 7. Quantitative check on 5×5 grids with loss $= 20$ and g $= 6$

degree	C	$p - \varepsilon$	900 frames	1800 frames	2700 frames	3600 frames
4	7	0.08	≥ 0.53	≥ 0.78	≥ 0.90	≥ 0.95
6	9	0.04	≥ 0.39	≥ 0.61	≥ 0.76	≥ 0.84
8	11	0.02	≥ 0.25	≥ 0.44	≥ 0.58	≥ 0.69

the system may get out of sync also for high values of the guard time g. The result of the quantitative check is reported in Table 6. The value p represents the center of the confidence interval computed by UPPAAL.

The compositional reasoning on execution modules adopted for cliques can be easily generalized to grid topologies. Table 7 fixes loss $= 20$ and g $= 6$. It reports lower bounds to the probability of getting out of sync within 900, 1800, 2700 and 3600 time frames. As said before, in the real implementation a time frame lasts for around $1sec$. Thus, when considering g $= 6$ and a message loss of 20%, we observe that the desync probability exceeds 0.5 in less than $15min$ for degree 4, in less than $30min$ for degree 6, and in less than $45min$ for degree 8. Table 7 outlines also how the performances of the protocol depend on the node degree: the probability of getting out of sync decreases for grid topologies with higher node degree.

Finally, let us give a taste of what happens when g $= 7$. Among the results on Table 6 we extend the case of degree 4 and loss $= 20$, where the probability of getting out of sync within 100 time frames lays in the interval $[0.03\,,\,0.09]$. The projection to 2700 time frames says that the probability of getting out of sync becomes greater than 0.54. In the real settings, this means that the probability of getting out of sync exceeds 0.5 in less than $45min$.

In conclusion, in the case of lossy communication, *small grid topologies have a high probability of getting out of sync even for high values of the guard time* g *and for low values of the loss probability. Moreover the probability of getting out of sync increases when decreasing the maximum node degree.*

5 Conclusions, Future and Related Work

Our work has been strongly inspired by a recent analysis [8] of the gMAC synchronization protocol on clique and line topologies, in the ideal case of non-lossy communication. That analysis provides constraints on the protocol parameters that are both necessary and sufficient for the correctness of the protocol for cliques of arbitrary size. Here we have carried on the work of [8] in the case of lossy communication. We have extended their model and obtained a network of probabilistic timed automata [6] which has been used for doing Statistical Model Checking within the UPPAAL toolset [7]. As a main result, we have showed that in the presence of message loss the constraints provided in [8] may be not sufficient to ensure clock synchronization of cliques of arbitrary size. Then, we have extended our analysis of the protocol to small grid topologies and again found that, in the case of lossy communication, the nodes of the grid may get out of sync with high probability. More interestingly, grid topologies with higher node degree have a smaller probability of desynchronization. This lets us to conjecture that higher connectivity helps synchronization protocols. In this respect, among the regular topologies, clique topologies are those with the best performances!

As in [8] we have assumed a fixed slot allocation. However, the implementation of gMAC includes a probabilistic dynamic slot allocation algorithm. The only analysis we are aware of the probabilistic gMAC algorithm appears in [16]. In that paper, mobile sensors do not use a fixed schedule to control medium access but instead employ gMAC's full decentralized slot allocation: gossiping is introduced to allow each node to decide when to send. Paper [16] analyzes the energy-efficiency of gMAC under the assumption of perfect clock synchronization. The protocol, formalised in the MoDeST language [4], is evaluated using the discrete-event simulator of the Möbius tool suite. We are planning to study the performance of the gMAC protocol with dynamic slot allocation in the case of lossy communication and realistic clock. In doing that, we intend to adopt either a (truncated) normal distribution or a (truncated) exponential distribution for modeling a more realistic delay between consecutive ticks.

Statistical Model Checking allows us to study networks of bigger size with respect to the state-of-the art model checking technology, such as PRISM [9,11]. SMC can be seen as a trade off between testing and formal verification: its approach consists in performing an appropriate number of simulations which are elaborated with statistical algorithms to verify if a given property is satisfied with a certain probability. Unlike an exhaustive approach, a simulation-based solution does not guarantee a correct result with a 100% confidence. It is only possible to bound the probability of making an error. In order to study bigger systems with an higher confidence, paper [5] proposes a distributed implementation of UPPAAL SMC by means of a master/slave architecture where several computers are used to generate simulations and a single master process is used to collect those simulations and perform the statistical test. We are planning to employ this approach to extend the confidence of the results we obtained in this paper.

Acknowledgments. The anonymous referees provided insightful comments.

References

1. Assegei, F.: Decentralized frame synchronization of a TDMA-based wireless sensor network. Master's thesis, Eindhoven University of Technology, Department of Electrical Engineering (2008)
2. Behrmann, G., David, A., Larsen, K.G.: c. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer (2004)
3. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST 2006. pp. 125–126. IEEE Computer Society (2006)
4. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MODEST: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. 32(10), 812–830 (2006)
5. Bulychev, P.E., David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Checking and distributing statistical model checking. In: Goodloe, A., Person, S. (eds.) NFM 2012. LNCS, vol. 7226. Springer (2012)
6. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical model checking for networks of priced timed automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer (2011)
7. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer (2011)
8. Heidarian, F., Schmaltz, J., Vaandrager, F.W.: Analysis of a clock synchronization protocol for wireless sensor networks. Theor. Comput. Sci. 413(1), 87–105 (2012)
9. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer (2011)
10. Nordic Semiconductors: nRF2401 Single-chip 2.4GHz Transceiver Data Sheet (2002)
11. Norman, G., Parker, D., Sproston, J.: Model checking for probabilistic timed automata. Formal Methods in System Design (2012), to appear
12. QUASIMODO: Preliminary description of case studies, deliverable 5.2 from the FP7 ICT STREP project 214755 (January 2009)
13. Sundararaman, B., Buy, U., Kshemkalyani, A.D.: Clock synchronization for wireless sensor networks: a survey. Ad Hoc Networks 3(3), 281–323 (2005)
14. van Vessem, I.: WSN gMAC protocol specifications. Tech. rep., CHESS B.V., Haarlem, NL (2008), version 1.1. Patent pending US 12 / 250,040
15. Wu, Y.C., Chaudhari, Q.M., Serpedin, E.: Clock synchronization of wireless sensor networks. IEEE Signal Process. Mag. 28(1), 124–138 (2011)
16. Yue, H., Bohnenkamp, H.C., Katoen, J.P.: Analyzing energy consumption in a gossiping mac protocol. In: Müller-Clostermann, B., Echtle, K., Rathgeb, E.P. (eds.) MMB&DFT 2010. LNCS, vol. 5987, pp. 107–119. Springer (2010)
17. Zhao, J., Govindan, R.: Understanding packet delivery performance in dense wireless sensor networks. In: Akyildiz, I.F., Estrin, D., Culler, D.E., Srivastava, M.B. (eds.) SenSys 2003. pp. 1–13. ACM (2003)

A New Representation of Two-Dimensional Patterns and Applications to Interactive Programming

I.T. Banu-Demergian[1], C.I. Paduraru[1], and G. Stefanescu[1]

Department of Computer Science, University of Bucharest, Bucharest, Romania
th_iulia84@yahoo.com, ciprian.paduraru2009@gmail.com,
and gheorghe.stefanescu@fmi.unibuc.ro

Abstract. Regular expressions and the associated regular algebra provide a rich formalism for specifying and analysing sequential models of computation. For parallel computation, extensions to handle two-dimensional patterns are often required. In this paper we present a new type of regular expressions for two-dimensional patterns based on contours and their composition. Targeted applications comes from the area of modelling, specification, analysis and verification of structured interactive programs via the associated scenario semantics.

Keywords: regular expressions, two-dimensional patterns, contours, structured interactive programming, formal methods

1 Introduction

Regular expressions and the associated regular algebra provide a rich formalism for specifying and analysing sequential models of computation. They were originally introduced by Kleene [17] in connection with neural networks and finite automata - Kleene theorem states that finite automata and regular expressions are equivalent (i.e., they specify the same language). In the meantime, regular expressions became a core formalism for many other models used in computer science. In particular, they provide the backbone of a rich algebraic theory of automata, see, e.g. [28,11,20,18,6,19,8].

For parallel computation, enrichment of the sequential models with mechanisms for modelling process interaction are needed. We only mention a Kleene theorem for Petri nets [27,13]: Petri nets and a class of concurrent regular expressions are equivalent. The result is based on the following procedure: (1) decompose the behaviour to have separate components where each transaction has no more than one input and one output place; (2) decompose the behaviour of a component to have an independent run for each initial token; (3) use the classical Kleene theorem for these sequential runs; (4) use synchronization and renaming to force the composition of these separate projected runs to behave as a run of the initial overall system. However, as it was often noticed (see, e.g., [18]), renaming has bad algebraic properties and should be avoided.

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 183–198, 2013.
DOI: 10.1007/978-3-642-40213-5_12,
© IFIP International Federation for Information Processing 2013

A natural semantics for parallel computation is provided by a kind of two-dimensional patterns/languages - for instance, messages sequence charts or scenarios fall into this category. A robust class of "regular two dimensional languages" has been identified in 1990's [14,22]; it may be specified by many equivalent formalisms, in particular by a 2-dimensional version of regular expressions 2RE, including intersection and renaming.

In this paper we present a new type of regular expressions for two-dimensional patterns n2RE based on contours and their composition. It avoids the use of intersection and renaming, being closer in spirit with classical 1-dimensional regular expressions. In this approach, the magic way of getting the intended language by renaming and intersection is replaced by a steady work of tiling shapes to build up the words step by step.

Our targeted applications comes from the area of modelling, specification, analysis and verification of structured interactive programs via the associated scenario semantics. Interactive computation [15] is becoming more and more important in the recent years, in particular due to the advance of multicore computation. We use a model rv-IS [35] based on space-time duality. In particular, finite interactive systems [34] are the space-time invariant extension of finite automata in this context. Agapia programming [12] is a core interactive programming language based on this model. We use the n2RE expressions to present a relational semantics for Agapia programs; it may be seen as an extension to two dimensions of the classical relational semantics of sequential computing models [23,24].

The paper is organized as follows. Section 2 presents a known approach using two sets of regular algebra operators, intersection, and renaming. Section 3 presents the new approach based on contours and Section 4 shows an application for getting a relational semantics for structured interactive programs. Related and future works and references conclude the paper.

2 A Known Approach

2.1 Finite Interactive Systems (FIS's) and Regular Expressions (2RE's)

Definition 1. A *finite interactive system (FIS)* [34,35] is defined by

- two types of nodes: *states* (denoted by numbers $1, 2, ...$) and *classes* (denoted by capital letters $A, B, ...$);
- transactions: $(A, 1) - a \rightarrow (B, 2)$, where a is a letter of the considered alphabet and $A, B, 1, 2$ are as above;
- specification of the *initial/final* states and classes. □

A useful *cross/tile representation* may be used; is is based on showing the transitions and stating which states and classes are initial/final. An example is

$S1$: 1 / A a B / 2   2 / A c A / 2   1 / B b B / 1   with $1, A$ initial and $2, B$ final.

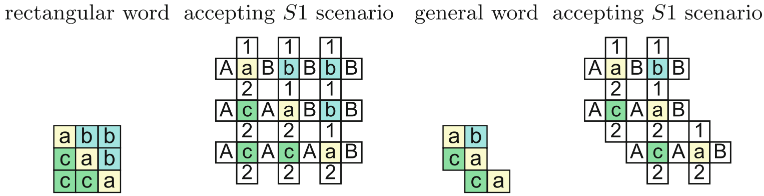


Fig. 1. FIS recognizing procedure

FIS recognizing procedure. The *FIS recognizing procedure* is via accepted scenarios. A *scenario* alternates class/state information and letters according to the FIS transitions. It is an *accepting scenario* if the northern border has initial states, the western border has initial classes, the eastern border has final classes, and the southern border has final states.

Graphically, a scenario may be easily obtained using the crosses representing the transitions and identifying the matching classes or states of the neighbouring cells. In Fig. 1 we show a few examples of scenarios for the FIS $S1$ above. Notice that the recognizing procedure may be applied to non-rectangular words, as well.

Definition 2. First, *simple 2-dimensional regular expressions* (simple 2RE's) are defined by two sets of regular operators (one for the vertical, the other for the horizontal direction) which share the additive part. Formally, they use:

1. the *additive operators*: `0` (for *empty set*) and `+` (for *union*);
2. the *vertical composition operators*: `I_v` (*vertical identity*), `;_v` (*vertical composition*) and `*_v` (*iterated vertical composition*); our preferred textual notation is: `|`, `;` and `*`;
3. the *horizontal composition operators*: `I_h` (*horizontal identity*), `;_h` (*horizontal composition*) and `*_h` (*iterated horizontal composition*); our preferred textual notation is: `-`, `>` and `^`.

Next, *2-dimensional regular expressions (2RE's)* are obtained adding intersection and renaming to simple 2RE's. Formally, they use the following additional operators

1. *intersection*: our preferred textual notation is `/\`;
2. *renaming* via a letter-to-letter homomorphism `rho:V->V'` (`V` and `V'` are the old and the new vocabulary, respectively). □

Examples. A few examples of 2RE expressions and typical specified words are presented in Fig. 2. They are related to the following expression

```
E = ( b* ; a ; c*)^ /\ (c^ > a > b^)*.
```

In Fig. 2.(1)–(4), typical words generated by the following expressions are presented: `b* ; a ; c*`; `( b* ; a ; c*)^`; `c^ > a > b^`; and `(c^ > a > b^)*`. It can be proved that the intersection `(b* ; a ; c*)^ /\ (c^ > a > b^)*` has only square words with `a` on the diagonal, `b` on the top right area and `c` on the

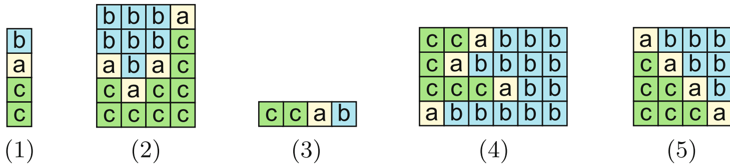


Fig. 2. A 2RE expression for the FIS S1

bottom left area, see Fig. 2.(5). The first part of the expression constrains the column patterns, while the second the rows. Intersection does the magic action of selecting only the square 2-dimensional words.

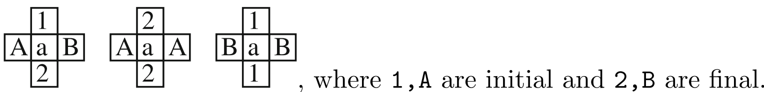
Theorem 1. *(connecting 2RE's to FIS's [14, 34, 35, 29]) The languages represented by finite interactive systems (FIS's) and those specified by 2-dimensional regular expressions (2RE's) are the same.*

Proof. (Sketch): As usual, more complicate is the passing from FIS's to 2RE's. It is done in two steps:

- for a FIS `S` with transitions having distinct letters the procedure is:
  - take a usual regular expression `Es` for the state-projected nondeterministic finite automaton (NFA) of `S` and another one `Ec` for the class-projected NFA of `S` (these NFA's are obtained from `S` ignoring one dimension)
  - an expression for `S` is `(Es)^/\(Ec)*`
- for an arbitrary FIS proceed as follows: first rename the transitions with new letters to apply the above step; then, apply the previous result; finally, use the renaming operator to rename back the new letters with their original version in the resulting expression.

For the other part, one can use the result in [29] showing that FIS's are equivalent with tile systems, then the relation between 2RE's and tile systems. □

Example: Let us consider the FIS `S2` defined by



, where `1,A` are initial and `2,B` are final.

The procedure is the following:

1. rename the 2-nd `a` as `c` and the 3-rd `a` as `b` to get different letters for transitions; actually, this way we get the FIS `S1` above;
2. get a 2RE for this new FIS; using the projected NFA's such an expression is

```
E1 = ( b* ; a ; c*)^ /\ (c^ > a > b^)*
```

3. rename back to get an expression for `S2`

```
E2 = rho [( b* ; a ; c*)^ /\ (c^ > a > b^)*]
```

   with `rho` mapping `a,b,c` into `a,a,a`, respectively.

Problems. There are a few problems with this approach, the main critics being the following:

- Intersection is a nonintuitive operator: Indeed, it is difficult to grasp what you get by intersecting two or more languages.
- The formalism is not robust under renaming: As an example, notice that the expression `E3 = ( a* ; a ; a*)^/\ (a^ > a > a^)*`, obtained by syntactically renaming `a,b,c` as `a` into the expression `E` above, represents all rectangular words of `a`'s, not only the square ones as one expects.
- Renaming is yet a still more nonintuitive operator: It's like writing in Chinese and getting an English text using a letter-to-letter morphism, losing most of the information.

The solution we propose to the above problems is the following:

- Construct a formalism for handling words of arbitrary shapes in the 2-dimensional plane;
- Introduce a powerful set of composition operators for these shapes (extending vertical/horizontal compositions and their iterated versions)

In other word, **the magic way of getting the intended language by renaming and intersection is to be replaced by a steady work of tiling shapes to get the words step by step**.

3 A New Approach

3.1 General 2-Dimensional Words

A *(pointed) contour* is a closed line, with a chosen starting point, on a rectangular grid $\mathbb{Z}\times\mathbb{Z}$ that divide the space into two disjoint regions: the internal area (which is required to be finite) and the external area. It will be represented using a sequence of letters from the set {`u,d,l,r`} (`u` stands for "up", `d` for "down", `l` for "left", and `r` for "right") and a placement (x, y) of the starting point. For simplicity, by default one can consider the starting point to be $(0, 0)$.

A few examples of contours are shown in Fig. 3. A representation for `C1` is `l_1 u_1 l_1 d_1 l_1 u_2 r_3 d_2`. The numbers after the letters are used to count repetitions. The contour starts at the chosen (black dot) point and travel clockwise. The interior area of a contour is the dashed (yellow) one. As the contour is surrounded clockwise, the area on the right is internal, while the one on the left is external. By changing the starting point, the representation is shifted circularly; for instance, `C2` is represented by `l_1 d_1 l_1 u_2 r_3 d_2 l_1 u_1`. Two slightly more complicate contours are shown in `C3` and `C4` in Fig. 3. As the last example shows, one can have contours with distinct disjoint components in its internal area, connected via lines travelled forth and back in the representation. The lines travelling into the internal areas are also called *tunnels*, while those sitting in the external areas *bridges*. As one can see, the tunnels and the bridges have a lot of freedom regarding both their forms and their physical placement;

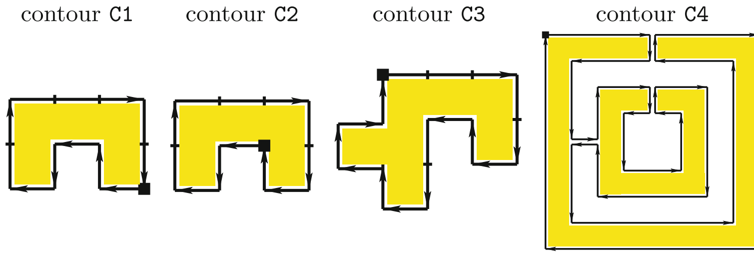


Fig. 3. Contours

for instance, they may have branches going nowhere. From the 2-dimensional words point of view, all these representations of contours are equivalent.

The formal definition of a contour requires quite a lot of preparation and it is presented below. A *closed line* `C` is a string over the set {`u`,`d`,`l`,`r`}, obeying the conditions `no(C,u) = no(C,d)` and `no(C,r) = no(C,l)`, where `no(C,x)` denotes the number of occurrences of `x` in `C`.

For a point $p = (x, y) \in \mathbb{Z}^2$, p_C^k specifies the information that C passes exactly k time through p; notice that $k \geq 0$. A vertical line segment $((x, y), (x, y + 1))$ is specified by its middle point $l = (x, y + 0.5)$; the notation l_C^k specifies that the difference between the "up" and the "down" times C passes through this segment is k; notice that $k \in \mathbb{Z}$ may be both positive or negative. Similarly, for an horizontal line segment $((x, y), (x + 1, y))$, denoted $l = (x + 0.5, y)$, the notation l_C^k says that k is the difference between the "right" and the "left" times C passes through l. Finally, a unit cell with the corners $\{(x, y), (x + 1, y), (x + 1, y + 1), (x, y + 1)\}$ is specified by its center point $c = (x + 0.5, y + 0.5)$.

For a cell $c = (x + 0.5, y + 0.5)$, the notation $c_{C,w}^k$ specifies how c is seen in C from a western perspective. Formally, let $z = max\{w \in \mathbb{Z} : w \leq x$ and $l = (w, y + 0.5)$ is such that l_C^k is true with $k \neq 0\}$; then $c_{C,w}^k$ is true if l_C^k is true for the line $l = (z, y + 0.5)$. In words, starting from the center of the cell and travelling horizontally towards the west there is a first line crossed and having a unequal up/down passings and, moreover, the difference between the "up" and the "down" passings along that line is k. The notations $c_{C,e}^k$, $c_{C,n}^k$, and $c_{C,s}^k$ are similarly introduced for the eastern, northern, and southern directions.

A cell is seen as *internal* if $c_{C,w}^k$ and $k > 0$. For the *external* property the condition is slightly different: either $c_{C,w}^k$ and $k < 0$ (i.e., going horizontally towards west, the there is a first line crossed with more down than up passings) or *there is no line crossed with unequal up/down passings.* This additional condition is needed to ensure the internal area of a valid contour is finite; for instance, `drul` is not a valid contour (see below the formal definition of valid contours).

With these notations, the correctness criteria for a string to represent a valid contour are the following: a closed line C represents a *valid contour* if:

- each cell is either internal from all directions, or external from all directions;
- for the internal cells, the conditions $c_{C,w}^k$, $c_{C,e}^k$, $c_{C,n}^k$, and $c_{C,s}^k$ are all satisfied with $k = 1$.

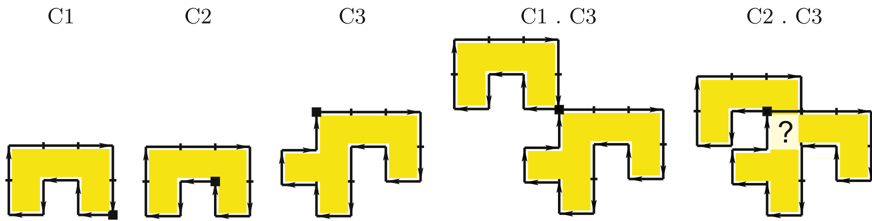


Fig. 4. Contours composition

The last condition is needed to avoid overlapping by multiple surroundings of the same internal area; for instance `rdlurdlu` is not valid, while `rdlu` is.

It is possible to replace the conditions here with conditions on the string itself, rather than on the lattice cells. However, such an approach is less intuitive (it is based on forbidden string configurations) and the details are quite complex; see [3].

A *general 2-dimensional word* is specified by a contour and a filling of its internal area with letters from the given alphabet. In the following we will mostly ignore this additional information as most of the difficulties are posed by the handling of the contours/shapes and not by the contents of their internal areas.

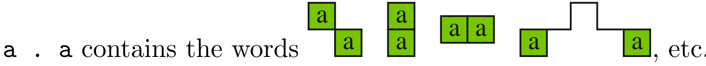
3.2 General Composition

A *general composition* operator '.' on contours may be defined as follows: given two contours, get a new contour by putting them together and identifying their starting points (the black dots). This means, one has to travel along the first contour and when he arrives back to the starting point, to travel along the next. In the string representation of the contours, the operation actually is concatenation `C1 . C2 = C1 C2`.

Comments: The condition to have a definite composite is to have a valid non-overlapping contour after the concatenation of the representations of the given contours (for instance, a pointed contour cannot be composed with itself). In particular, this implies that there is no constraint on the contents of the internal areas of the contours, hence the operation is straightforwardly extended to general 2-dimensional words. This is a very powerful and general composition operator, indeed.

For a graphical example, C1 . C3 in Fig. 4 shows a composition leading to a valid contour, while C2 . C3 leads to a string representation which does not represent a valid contour (it has overlapping areas).

This composition is extended to two-dimensional words as follows. For two words `W1, W2`, consider arbitrary contours `C1, C2` representing them (having as internal areas the shapes of the words) and arbitrary positions as starting points of these contours. Then, `W1 . W2` consists of all words resulting from valid compositions of such contours and placing the letters of the words `W1, W2`

in the corresponding positions of the resulting composites. E.g., the composite `a . a` contains the words , etc.

3.3 Particular Composition Operators

The new type of 2-dimensional regular expressions, to be defined below, put constraints on the contact elements of the composed words. These constraints acts on the following three types of elements: *side borders*, *land corners* (turning points on the contour having 3 neighbouring cells outside the word and one neighbouring cell inside), and *golf corners* (turning points on the contour with at least 2 neighbouring cells inside and one neighbouring cell outside). An example is shown in Fig. 5(1). The resulting restricted composition operators extend the usual vertical and horizontal composition operators used on rectangular words.

Points of interest on the words borders. Let us use the following notation (their meaning is explained right after the listing):

- *side borders*: elements in `C1={w,e,n,s}`, where `w` stands for "west border", `e` for "east border", `n` for "north border", and `s` for "south border";
- *land corners*: elements in `C2={nw,ne,sw,se}`, where `nw` stands for "north-west land corner", `ne` for "north-east land corner", `sw` for "south-west land corner", and `se` for "south-east land corner";
- *golf corners*: elements in `C3={nw',ne',sw',se'}`, where `nw'` stands for "north-west golf corner", `ne'` for "north-east golf corner", `sw'` for "south-west golf corner", and `se'` for "south-east golf corner".

A line $l = (x, y+0.5)$ is on the *east border* of a word f if the cell $c = (x-0.5, y+0.5)$ is in the internal area of f, while the cell $c = (x+0.5, y+0.5)$ is in the external area of f. For the other west, north, and south directions, the definition is similar. A point $p = (x, y) \in \mathbb{Z}^2$ is on the *south-east land corner border* of a word f if the cell $c = (x-0.5, y+0.5)$ is in the area of f, while the other 3 cells around are not in the area of f (they are in the external area of f). For the other 3 types of land corners the definition is similar. A point $p = (x, y) \in \mathbb{Z}^2$ is on the *south-east golf corner border* of a word f if the cell $c = (x-0.5, y+0.5)$ is not in the area of f (it is in the external area of f), while at least 2 of the other 3 cells around are in the area of f. For the other 3 types of golf corners the definition is similar.

Glueing combinations. The constraints on glueing the borders of the words in the composite word are independently put on one or more of the following combinations (x, y):

- x and y are different and either they are both in $\{e, w\}$, or both in $\{s, n\}$, or both are land corners in $\{nw, ne, sw, se\}$, or both are combinations golf-land corners for the same directions.

Spelling out the resulting combinations we get the following lists:

– linking side borders: `L1={(w,e),(e,w),(n,s),(s,n)}`;
– linking land corners: `L2={(nw,ne),(nw,se),(nw,sw),(ne,nw),(ne,se),(ne,sw),(se,nw),(se,ne),(se,sw),(sw,nw),(sw,ne),(sw,se)}`;
– linking golf-land corners: `L3={(nw',nw),(nw,nw'),(ne',ne),(ne,ne'),(se',se),(se,se'),(sw',sw),(sw,sw')}`.

The set of all combinations in $L1 \cup L2 \cup L3$ is denoted by `Connect`.

Constricting formulas. On each of the above eligible glueing combination (x, y) we put a constrain given by a propositional logic formula[1] $F \in PL(\phi_1, \phi_2, \phi_3, \phi_4)$, i.e., a boolean formula built up starting with the following atomic formulas:

$$\phi_1(x, y) = \text{“}x < y\text{”}, \phi_2(x, y) = \text{“}x = y\text{”}, \phi_3(x, y) = \text{“}x > y\text{”}, \phi_4(x, y) = \text{“}x \# y\text{”}.$$

The meaning of the connectors is the following: “<” - left is included into the right; “=” - left is equal to the right; “>” - left includes the right; “$x \# y$” - left and right overlaps, but no one is included in the other.

For instance: $f(e = w)g$ means “restrict the general composition of f and g such that the east border of f is identified to the west border of g”; $f(e > w)g$ - the east border of f includes all the west border of g, but some east borders of f may still be not covered by west borders of g; etc.

We also use the notation

$$\phi_0(x, y) = \text{“}x \ ! \ y\text{”, where “!” means empty intersection.}$$

Actually, this is a derived formula $\neg(\phi_1(x, y) \vee \phi_2(x, y) \vee \phi_3(x, y) \vee \phi_4(x, y))$.

Particular composition operators. We are now in a position to introduce the particular composition operators induced by the above constricting formulas.

Definition 3. (restricted compositions) A *restriction formula* ϕ is a boolean combination in $PL(F_1, \ldots, F_n)$, where F_i are constricting formulas involving certain eligible glueing combinations $(x_i, y_i) \in Connect$. A *restricted composition operation* $_(F)_$ is the restriction of the general composition to composite words satisfying F. A word $h \in f \ . \ g$ belongs to $f \ (F) \ g$ if for all glueing combinations (x_i, y_i) occurring in F the contact of the x_i border of f and y_i border of g satisfies F_i. □

This interpretation shows the constricting formulas act on the involved glueing combinations, while for the glueing combinations (x_j, y_j) not occurring in the formula no constraints are imposed, at all. Other default conventions are possible, too; for instance stating that what is not specified should not touch.

Notice that the restricted composition operations are not always associative; e.g., `((a (s=n) a) (e>w) b) (e>w) c` $\neq$ `(a (s=n) a) (e>w) (b (e>w) c)`. When some parentheses are missing, we suppose a left-parentheses order applies, as in ((C1 op C2) op C3).

[1] $PL(Atom)$ denotes the set of propositional logic formulas built up with atomic formulas in *Atom*. For typing reasons, the boolean operations “not”, “and”, and “or” are denoted by “!”, “&”, and “|”, respectively.

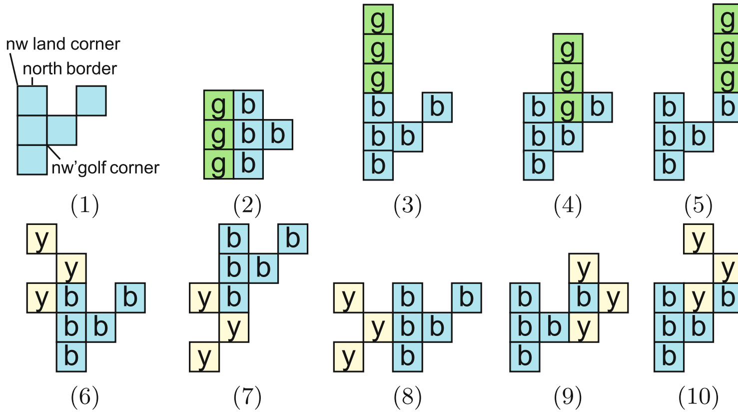


Fig. 5. Particular composition operators

Examples. A few examples are shown in Fig. 5. Let `g`, `b`, and `y` represent the green, blue, and yellow areas, respectively. Then, {(2)} (the word described in Fig. 5(2)) is the result of `g (e=w) b`. Similarly: `g (s< n) b` is the set of words {(3),(4),(5)}; `b (ne<ne') y` is {(9)}; `y (s#n & w#e) b` is {(9),(10)}. The words for `y (e#w) b` strictly include the set {(6),(7),(8),(10)}; one can use the expression `y (e#w & ne!nw & se!sw) b` to exclude a few words from `y (e#w) b` and to get precisely the set {(6),(7),(8),(10)}.

Definition 4. (iterated composition operators) The *iterated composition operators* are denoted by $*(F)$, for a restriction formula F. □

Definition 5. The set of expressions obtained using the operators defined so far are denoted by n2RE's; they represent our *new type of regular expressions for two-dimensional patterns/words.* □

Examples, related to S1. The examples in Fig. 6 are related to $S1$, the original FIS we have considered in the beginning of the section. We first show the expressions, then include samples of typical words associated to these expressions. Combined with the constraint to have rectangular words, the final regular expression `Eabc` specifies the language of `S1`.

Observation. The formalism is robust, in particular it commutes with renaming: renaming letters either in the associated words or in the given expression leads to the same set of general 2-dimensional words.

4 A Relational Semantics for Structured Interactive Programs

In this section we show how the introduced regular expressions can be used to get a relational semantics for structured interactive programs presented in

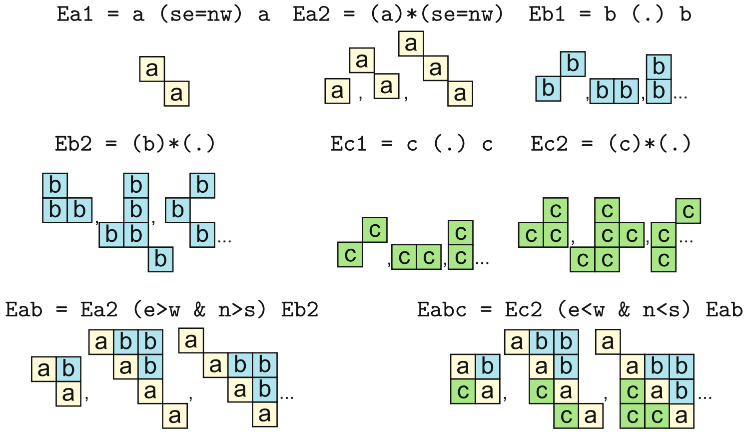


Fig. 6. A n2RE expression for the FIS $S1$

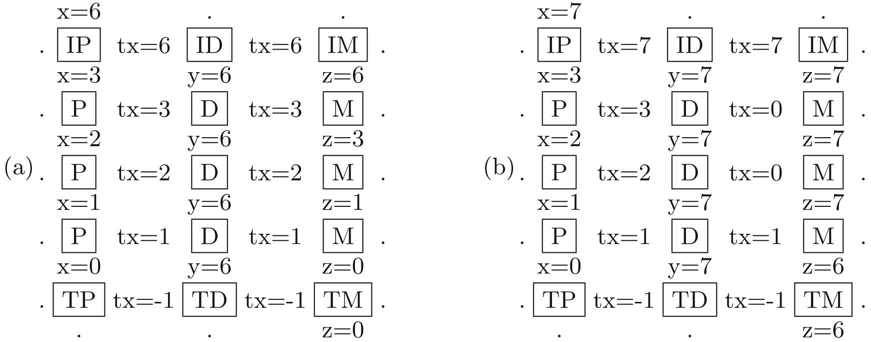


Fig. 7. Two scenarios for computing perfect numbers

the rv-IS formalism [35,12]. The operational semantics of structured programs is given in terms of scenarios. In Fig. 7(a) we illustrate an rv-IS scenario for deciding whether the number 6 is a perfect number (i.e., it is equal to the sum of its proper divisors); in (b) it is a scenario for testing if 7 is a perfect number.

In this representation, the actions to be performed are placed in the square cells. One example is the cell with the identifier `P` placed in the 2nd row and the 1st column in (a). On the top and the bottom of this cell there is the same state variable `x` with its concrete values 3 and 2. Actually, the effect of `P` on the memory state is to decrease `x` by 1. With respect to the interaction part, this cell has no variables for its left border (a fact specified by the '.' inserted there) and has a variable `tx` at its right border. The effect of `P` on the interaction part is to set in `tx` the input value 3 of `x` in order to be used in other columns/processes.

In Fig. 8 we present relational specifications for the cells used in Fig. 7. All these cells have functional behaviour, hence the corresponding relations may be specified as partial functions in the following way:

```
Cell(west,north) = (east,south), if Condition.
```

4.1 Example - Imperative Programming Style

We start with the following expression specifying scenarios checking if a number n is perfect

```
      [((IP (e=w) ID) (e=w) IM)
(s=n) (((P (e=w) D) (e=w) M) *(s=n))]
(s=n) ((IP (e=w) ID) (e=w) IM)
```

In this model we can imagine that we have three processes: one generates all the numbers in the set $\{n/2, \ldots, 1\}$ (with module `P`), one checks if a number is a divisor of n (module `D`) and the last one updates a variable z (module `M`). Modules `IP`, `ID` and `IM` are used for initializations and `TP`, `TD` and `TM` for termination. At the end of the program, if the variable z is 0, then the number n is perfect.

In order to show how we can construct a scenario using the expression above let us consider a concrete example for $n = 6$. The scenario for $n = 6$ is presented in Fig. 7 (a).

```
 IP((.),(x)) = ((tx'),(x')) with tx' = x and x' = x/2; defined if x ≥ 2
ID((tx),(.)) = ((tx'),(y')) with tx' = tx and y' = tx
IM((tx),(.)) = ((.),(z'))   with z' = tx
  P((.),(x)) = ((tx'),(x')) with tx' = x and x' = x-1; defined if x > 0
 D((tx),(y)) = ((tx'),(y')) with y' = y and
                            tx' = if(y%tx=0) then tx else 0
 M((tx),(z)) = ((.),(z'))   with z' = z-tx
 TP((.),(x)) = ((tx'),(.))  with tx' = -1; defined if x = 0
TD((tx),(y)) = ((tx'),(.))  with tx' = -1
TM((tx),(z)) = ((.),(z'))   with z' = z
```

Fig. 8. Relational semantic specifications for the cells used in Fig. 7

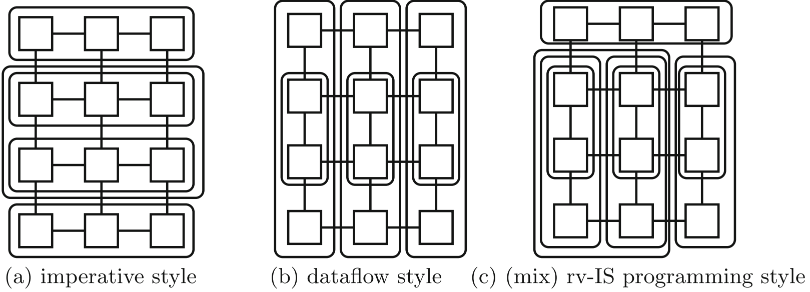
In the first line of the scenario we initialize the processes with the needed informations: module `IP` is reading the value $n = 6$ and provides the first process with $x = 3$ and declare a temporal variant of n, namely $tn = 6$, that will be used by modules `ID` and `IM` for the other initializations; modules `ID` and `IM` use the temporal variable tn for initializing the other two processes with the initial value of n, namely $y = 6$, $z = 6$, respectively.

In the next step, module `P` produces a temporal data $tx = 3$ (tx is equal with the data x of the first process) and decrease x. Module `D` verifies if tx is a divisor of y and, if no, it resets the value of tx to 0. Finally, module `M` decreases the value of z by tx. Notice that module `M` decreases the value of z only with the divisors of the initial variable n. We continue this steps until the variable x becomes 0.

A final line contains terminating modules that rearrange some interfaces, keeping only the relevant result z.

4.2 Dataflow and Mixed Imperative-Dataflow Programming Styles

The above model corresponds to the construction of scenarios by rows and it exhibits a (parallel) *imperative programming style*, illustrated in Fig. 9(a).



(a) imperative style (b) dataflow style (c) (mix) rv-IS programming style

Fig. 9. Programming strategies

The same computing scenarios may be generated in many other ways. Below is a model which constructs the same scenarios by columns, exhibiting a *dataflow computing style*, illustrated in Fig. 9(b):

```
      [((IP (s=n) (P *(s=n)) (s=n) TP)
(e=w) ((ID (s=n) (D *(s=n)) (s=n) TD)]
(e=w) ((IM (s=n) (M *(s=n)) (s=n) TM)
```

In this implementation, the processes corresponding to the 2nd and the 3rd columns act as "services": they receive initialization data (here the value of `n`), then a stream of data to act individually on each one according to the service function (for the 2nd process this function is the check for divisibility, while for the 3rd is subtraction), and finally a termination token (represented here by -1).

Finally, we present a last model, illustrated in Fig. 9(c), which *mixes the imperative and dataflow styles*

```
      ((IP (e=w) ID) (e=w) IM)
(s=n) [(((P (e=w) D) *(s=n)) (s=n) (TP (e=w) TD))
       (e=w) ((M *(s=n)) (s=n) TM)]
```

In this version the construction of the scenarios is as follows. It starts by constructing the 1st line of the scenarios. Then, the remaining parts of the first two columns are generating in the same way as with the initial model (that is, by an imperative style). Moreover, the same is done separately for the remaining part of the third column. Finally, these parts are composed horizontally (following a dataflow style).

5 Related and Future Works

Related work. Regular expressions are introduced in the seminal paper by Kleene [17] on the representation of events in neural nets and automata; it was published in the early 1950s. Kleene theorem (i.e., the equivalence between finite automata and appropriate regular expressions) was extended to cover other computing models of interest and is a basis for the development of algebraic theories for those models.

The algebraic theory of finite automata is based on semirings enriched with an axiomatic iteration operator; often the term Kleene algebra is used in this context. It was steadily developed since 1960s till now, including deep results as Krob's solution [19] for two deep conjectures of Conway [11]. We notice the interest in getting complete equational axiomatizations; see, e.g., [28,18,6,19,8]. A few books recording the results are [11] and [20].

When (matrices over) semirings are replaced by more general algebraic structures as symmetric (strict) monoidal categories, the iteration operators may have different expressive powers and axiomatisations. Trace monoidal categories [30,9,16,33] are now recognized as a powerful formalism for iterative processes, with wider applications than Kleene algebras; in particular they apply to circuits, dataflow computation, quantum computation, etc. Translations between various formalisms using axiomatic iteration operators may be found in [31,32,7,10,33]. Axiomatic iteration operators are also present in process calculi; a few papers are [25,5,26].

Parallel computation often requires the enrichment of the sequential computation models with mechanisms for modelling process interaction. We mention three examples of extensions of Kleene theorem into such a context: Kleene theorems for tile systems [14], for Petri nets [27,13], and for timed automata [1,2]. In all these contexts, the Kleene theorem is based on the following procedure: (1) decompose/project the behaviour to have separate sequential runs; (2) use the classical Kleene theorem for these sequential runs; (3) use synchronization and renaming to force the composition of these separate projected runs to behave as the initial overall system. It was noticed (see, e.g., [18]) that renaming has bad algebraic properties and should be avoided.

The study of two dimensional languages has started in 1960's; see [14,22]. In 1990's, a robust class of "regular two dimensional languages" has been identified; it may be specified either by tile systems, or by a type of cellular automata, or by a class of monadic second-order formulas, etc. Unfortunately, the class is quite complex - for instance, emptiness property is not decidable, see [21].

Interactive computation [15] is becoming more and more important in the recent years, in particular due to the advance of multicore computation. We use a model rv-IS [35] based on space-time duality. In particular, finite interactive systems [34] are the space-time invariant extension of finite automata in this context. A Kleene theorem for finite interactive systems follows directly from their equivalence with tile systems [29]. Agapia programming [12] is a core interactive programming language based on this model. The relational semantics described in the present paper for Agapia programs may be seen as an exten-

sion to 2 dimensions of the classical relational semantics of sequential computing models [23, 24].

Future work. There are many directions to continue the research presented in this paper. We are particularly interested to develop the theoretical basis of the model (e.g., to prove a Kleene theorem for finite interactive systems[2]; to look for an associated algebraic theory; etc.) and to provide a software tool for manipulating n2RE's. Among the possible applications we mention:

- the study of massively parallel, interactive OO-programs (semantics, specification, verification, etc.), in particular the programs written in the structured interactive programming language Agapia;
- applications to image processing, in particular learning n2RE as a image recognition procedure;
- modelling discrete physical or biological systems.

Acknowledgements. The research reported in this paper was partially supported by Deploy Project, an FP7 Integrated Project supported by European Commission (Grant No. 214158). We thank the anonymous reviewers for their suggestions for improving the presentation of the results.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM 49, 172–206 (2002)
3. Banu-Demergian, I., Stefanescu, G.: On the contour representation of two-dimensional patterns (2013), draft
4. Banu-Demergian, I., Stefanescu, G.: Representation of scenarios in finite interactive systems (2013), draft, submitted
5. Bergstra, J., Bethke, I., Ponse, A.: Process algebra with iteration. The Computer Journal 60, 109–137 (1994)
6. Bloom, S., Esik, Z.: Equational axioms for regular sets. Mathematical Structures in Computer Science 3, 1–24 (1993)
7. Bloom, S., Esik, Z.: Iteration Theories: The Equational Logic of Iterative Processes. Springer, Berlin (1993)
8. Bonsangue, M., Rutten, J., Silva, A.: A Kleene theorem for polynomial coalgebras. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 122–136. Springer, Heidelberg (2009)
9. Cazanescu, V., Stefanescu, G.: Towards a new algebraic foundation of flowchart scheme theory. Fundamenta Informaticae 13, 171–210 (1990)
10. Cazanescu, V., Stefanescu, G.: Feedback, iteration and repetition. In: Păun, G. (ed.) Mathematical Aspects of Natural and Formal Languages, pp. 43–62. World Scientific, Singapore (1995)

[2] Recently, the first and the last authors presented a characterization theorem for FIS languages in [4]. Their result shows that a slightly extended class of n2RE expressions and a mechanism for solving recursive equations suffice to represent FIS languages.

11. Conway, J.: Regular Algebra and Finite Machines. Chapman and Hall (1971)
12. Dragoi, C., Stefanescu, G.: Agapia v0.1: A programming language for interactive systems and its typing system. Electronic Notes in Theoretical Computer Science 203(3), 69–94 (2008)
13. Garg, V., Ragunath, M.: Concurrent regular expressions and their relationship to Petri nets. Theoretical Computer Science 96, 285–304 (1992)
14. Giammarresi, D., Restivo, A.: Two-dimensional languages. In: Handbook of Formal Languages, pp. 215–267. Springer (1997)
15. Goldin, D., Smolka, S., Wegner, P.: Interactive computation: The new paradigm. Springer (2006)
16. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. In. Proceedings of the Cambridge Philosophical Society, vol. 119 (1996)
17. Kleene, S.: Representation of events in nerve nets and finite automata. Automata Studies (34), 3 (1956)
18. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. In: LICS 1991, pp. 214–225. IEEE (1991)
19. Krob, D.: Complete systems of β-rational identities. Theoretical Computer Science 89, 207–343 (1991)
20. Kuich, W., Salomaa, A.: Semirings, automata and languages. Springer, Berlin (1985)
21. Latteux, M., Simplot, D.: Context-sensitive string languages and recognizable picture languages. Information and Computation 138(2), 160–169 (1997)
22. Lindgren, K., Moore, C., Nordahl, M.: Complexity of two-dimensional patterns. Journal of Statistical Physics 91(5–6), 909–951 (1998)
23. Maddux, R.: Relation-algebraic semantics. Theoretical Computer Science 160, 1–85 (1996)
24. Manes, E., Arbib, M.: Algebraic approaches to program semantics. Springer, Berlin (1986)
25. Milner, R.: Flowgraphs and flow algebras. Journal of the ACM (JACM) 26(4), 794–818 (1979)
26. Milner, R.: Action calculi V : Reflexive molecular forms (1994), draft, Department of Computer Science, University of Edinburgh
27. Petri, C.: Kommunikation mit automaten. Ph.D. thesis, Instituts fur Instrumentelle Mathematik, Bonn, Germany
28. Salomaa, A.: Two complete axiom systems for the algebra of regular events. Journal of the ACM (JACM) 13(1), 158–169 (1966)
29. Sofronia, A., Popa, A., Stefanescu, G.: Undecidability results for finite interactive systems. Romanian Journal of Information Science and Technology 12(2), 265–279 (2009), also: Arxiv, CoRR abs/1001.0143 (2010)
30. Stefanescu, G.: Feedback Theories (A Calculus for Isomorphism Classes of Flowchart Schemes). Preprint Series in Mathematics, vol. 24. INCREST (1986); also in: Revue Roumaine de Mathematiques Pures et Applique 35, 73–79 (1990)
31. Stefanescu, G.: On flowchart theories: Part I. The deterministic case. Journal of Computer and System Sciences 35(2), 163–191 (1987)
32. Stefanescu, G.: On flowchart theories: Part II. The nondeterministic case. Theoretical Computer Science 52(3), 307–340 (1987)
33. Stefanescu, G.: Network algebra. Springer (2000)
34. Stefanescu, G.: Algebra of networks: Modeling simple networks as well as complex interactive systems. In: Proof and System-Reliability, pp. 49–78. Springer (2002)
35. Stefanescu, G.: Interactive systems with registers and voices. Fundamenta Informaticae 73(1), 285–305 (2006)

Push-Down Automata with Gap-Order Constraints

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Giorgio Delzanno[2], and Andreas Podelski[3]

[1] Uppsala University
[2] University of Genova
[3] University of Freiburg

Abstract. We consider push-down automata with data (PDAD) that operate on variables ranging over the set of natural numbers. The conditions on variables are defined via gap-order constraint. Gap-order constraints allow to compare variables for equality, or to check that the gap between the values of two variables exceeds a given natural number. The messages inside the stack are equipped with values that are natural numbers reflecting their "values". When a message is pushed to the stack, its value may be defined by a variable in the program. When a message is popped, its value may be copied to a variable. Thus, we obtain a system that is infinite in two dimensions, namely we have a stack that may contain an unbounded number of messages each of which is equipped with a natural number. We present an algorithm for solving the control state reachability problem for PDAD based on two steps. We first provide a translation to the corresponding problem for context-free grammars with data (CFGD). Then, we use ideas from the framework of well quasi-orderings in order to obtain an algorithm for solving the reachability problem for CFGDs.

1 Introduction

Model checking has become one of the main techniques for algorithmic verification of computer systems. The original applications were found in context of finite-state systems, such as hardware circuits, where the behavior of the system can be captured by a finite state machine. In the last two decades, there has also been a large amount of work devoted to extending model checking so that its can handle models with *infinite* state spaces such as Petri nets, timed automata, push-down systems, counter automata, and channel machines. Recent works have considered systems that are infinite in *multiple* dimensions. For instance, many classes of timed protocols are *parameterized* (consist of unbounded numbers of components), and hence they can be naturally modeled by *timed Petri nets* [10]. Also, many message passing protocols have behaviors that are constrained by timing conditions, giving rise to *timed channel systems* [5].

In particular, Push-Down Automata (PDA) have been studied extensively as a model for the analysis of recursive programs (e.g., [12,33,23,25]). The model

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 199–216, 2013.
DOI: 10.1007/978-3-642-40213-5_13,
© IFIP International Federation for Information Processing 2013

of PDA has been extended to allow quantitative reasoning with respect to time [1] and probabilities [26,24]. However, all existing models assume finite-state control, which means that variables in the program are assumed to range over finite domains. In this paper, we consider an extension of PDA, which we call PDAD, that strengthens the model in two ways. First, in addition to the stack, a PDAD also operates on a number of variables ranging over the natural numbers. Furthermore, each message inside the stack is equipped with a natural number which represents its "value". Thus, we get a model that is possibly unbounded in two dimensions, namely we have an unbounded number of messages inside the stack each of which has an attribute that is a natural number. The operations allowed on the stack are the standard *push* and *pop* operations. However, when pushing a symbol to the stack, its value may be defined to be the value of a program variable. Also, when a message is popped, then its value may be copied to a variable. A PDAD allows comparing the values of variables according to the *gap-order* constraint system, where two variables may be tested for equality, or for checking that there is a minimal gap (defined by a natural number) between the values of the two variables. Also, a variable may be assigned a new arbitrary value, the value of another variable, or a value that is at least some (given) natural number larger than the value of another variable. In this manner, the model of PDAD subsumes two known models, namely that of PDA (which we get by removing the variables in the program and by neglecting the values of the symbols in the stack), and the model of *Integral Relational Automata* [15] (which we get by removing the stack).

In this paper, we show decidability of the control reachability problem for PDAD. Given a control (local) state of the automaton, we check whether the automaton reaches the state from its initial configuration. We solve the problem in two steps. We introduce a class of *Context-Free Grammars with Data* (CFGD). In a CFGD, each non-terminal has an arity. The grammar generates *terms* each of which is either a terminal or a non-terminal equipped with a tuple of natural number (as many as its arity). An application of a production rewrites a term to a *set* of terms. Such an application is constrained by the arguments of the involved non-terminals. The constraints are defined by gap-order conditions. For CFGD, we solve a reachability problem in which we ask whether it is possible to derive a set of terms each of which is a terminal belonging to a given set of terminals. In the first step of our method, we give a reachability analysis algorithm that solves the above mentioned problem for CFGDs.

The algorithm is based on a constraint representation of infinite sets of terms, and it is formulated within the framework of well structured transition systems [4,6].

The second step of our method translates a given PDAD into a CFGD so as to exploit the corresponding reachability analysis procedure to solve control state reachability for PDADs.

To our knowledge our result yields a new decidable fragment of pushdown automata with data (see Section 10).

2 Preliminaries

In this section, we introduce some notations and definitions that we will use in the rest of the paper. We use $\mathbb{N}$ to denote the set of natural numbers.

We fix a finite set $\mathcal{V}$ of variables that range over $\mathbb{N}$. A *valuation* is a mapping $Val : \mathcal{V} \to \mathbb{N}$, i.e., it assigns a natural number to each variable. Given a variable $x \in \mathcal{V}$, a natural number $c \in \mathbb{N}$, and a valuation $Val : \mathcal{V} \to \mathbb{N}$, we use $Val\,[x \leftarrow c]$ to denote the valuation Val' defined as follows: $Val'(x) = c$, and $Val'(y) = Val(y)$ for all $y \in (\mathcal{V} \setminus \{x\})$.

A *renaming* is a mapping $Ren : \mathcal{V} \to \mathcal{V}$, i.e., it renames each variable to another one. A renaming Ren does not need to be injective, i.e., several variables may be renamed to the same variable by Ren. We say that Ren is a renaming for W if $Ren\,(x) \in W$ for all $x \in \mathcal{V}$.

For a set A, we use A^* to denote the set of finite words over A. We use ϵ to denote the empty word. For words $\alpha_1, \alpha_2 \in A^*$, we use $\alpha_1 \cdot \alpha_2$ to denote the concatenation of α_1 and α_2.

A *transition system* is a tuple $\langle \Upsilon, \gamma_{init}, \longrightarrow \rangle$ where Υ is a (potentially infinite) set of configurations, $\gamma_{init} \in \Upsilon$ is the initial configuration, and $\longrightarrow \subseteq \Upsilon \times \Upsilon$ is the transition relation. As usual, we write $\gamma \longrightarrow \gamma'$ to denote that $\langle \gamma, \gamma' \rangle \in \longrightarrow$, and use $\xrightarrow{*}$ to denote the reflexive transition closure of $\longrightarrow$. For a configuration $\gamma \in \Upsilon$ and a set $\Gamma \subseteq \Upsilon$ of configurations, we use $\gamma \xrightarrow{*} \Gamma$ to denote that $\gamma \xrightarrow{*} \gamma'$ for some $\gamma' \in \Gamma$.

3 Push-Down Automata with Data

In this section, we introduce *Push-Down Automata with Data* (PDAD) that are extensions of the classical model of Push-Down Automata (PDA). First, we define the model, then we define the operational semantics, i.e., the transition system induced by a PDAD, and finally we introduce the reachability problem. As in the case of a PDA a PDAD operates on an unbounded stack to which it can push (append) messages and from which it can pop (remove) message in last-in-first-out manner. The messages are chosen from a finite alphabet. PDADs extend PDAs in two ways. First, in addition to the stack, the automaton is equipped with a finite set of variables ranging over natural numbers. Second, each message inside the stack is equipped by a natural number that represents its "value". The allowed operations on variables are defined by the *gap-order* constraint system [15,31]. More precisely, the model allows non-deterministic value assignment, copying the value of one variable to another, and assignment of a value v to some variable such that v is larger of at least a given natural number than the current value of another variable. The transitions may be conditioned by tests that compare the values of two variables for equality, or that give the minimal allowed gap between two variables. A *push* operation may copy the value of a variable to the pushed message, and a *pop* operation may copy the value of the popped message to a variable.

Model. A PDAD $\mathcal{A}$ is a tuple$\langle Q, q_{init}, A, \Delta \rangle$ where Q is the finite set of states, $q_{init} \in Q$ is the initial state, A is the stack alphabet, and Δ is the transition relation. We remark that the stack alphabet is infinite since it consists of pairs $\langle a, \ell \rangle$ where a is taken from a finite set and ℓ is a natural number. A transition $\delta \in \Delta$ is a triple $\langle q_1, op, q_2 \rangle$ where $q_1, q_2 \in Q$ are states and op is an *operation* of one of the following forms: (i) nop is an empty operation that does not change the values of the variables or the content of the stack, (ii) $x \leftarrow *$ assigns non-deterministically an arbitrary value in $\mathbb{N}$ to the variable x, (iii) $y \leftarrow x$ copies the value of variable x to y, (iv) $y \leftarrow (>_c x)$ assigns non-deterministically to y a value that exceeds the current value of x by c (so the new value of y is $> x + c$), (v) $y = x$ checks whether the value of y is equal to the value of x, (vi) $x <_c y$ checks whether the gap between the values of y and x is larger than c, (vii) $push\,(a)\,(x)$ pushes the symbol $a \in A$ to the stack and assigns to it the value of x, and (viii) $pop\,(a)\,(x)$ pops the symbol $a \in A$ (if a is the top-most symbol at the stack) and assigns its value to the variable x.

Transition System. A PDAD induces a transition system as follows. A *configuration* γ is a triple $\langle q, Val, \alpha \rangle$ where $q \in Q$ is a state, $Val : \mathcal{V} \mapsto \mathbb{N}$ is a valuation, and $\alpha \in (A \times \mathbb{N})^*$ defines the content of the stack (each element of the word is a pair $\langle a, c \rangle$ where a is the symbol and c is its value).

We define the transition relation $\longrightarrow := \cup_{\delta \in \Delta} \xrightarrow{\delta}$, where $\xrightarrow{\delta}$ describes the effect of the transition δ. For configurations $\gamma = \langle q, Val, \alpha \rangle$, $\gamma' = \left\langle q', Val', \alpha' \right\rangle$, and a transition $\delta = \langle q_1, op, q_2 \rangle \in \Delta$, we write $\gamma \xrightarrow{\delta} \gamma'$ to denote that $q = q_1$, $q' = q_2$, and one of the following conditions is satisfied:

- op is nop, $Val' = Val$, and $\alpha' = \alpha$. The values of the variables and the stack content are not changed.
- op is $x \leftarrow *$, $Val' = Val\,[x \leftarrow c]$ where $c \in \mathbb{N}$, and $\alpha' = \alpha$. The value of the variable x is changed non-deterministically to some natural number. The values of the other variables and the stack content are not changed.
- op is $y \leftarrow x$, $Val' = Val\,[y \leftarrow Val\,(x)]$, and $\alpha' = \alpha$. The value of the variable x is copied to the variable y. The values of the other variables and the stack content are not changed.
- op is $y \leftarrow (>_c x)$, $Val' = Val\,[y \leftarrow c']$, where $c' > Val\,(x) + c$, and $\alpha' = \alpha$. The variable y is assigned non-deterministically a value that exceeds the value of x by c. The values of the other variables and the stack content are not changed.
- op is $y = x$, $Val\,(y) = Val\,(x)$, $Val' = Val$, and $\alpha' = \alpha$. The transition is only enabled if the value of y is equal to the value of x. The values of the variables and the stack content are not changed.
- op is $x <_c y$, $Val\,(y) > Val\,(x) + c$, $Val' = Val$, and $\alpha' = \alpha$. The transition is only enabled if the value of y is larger than the value of x by more than c. The values of the variables and the stack content are not changed.
- op is $push\,(a)\,(x)$, $Val' = Val$, and $\alpha' = \langle a, Val\,(x) \rangle \cdot \alpha$. The symbol a is pushed onto the stack with a value equal to that of x.

– *op* is $pop(x)(a)$, $\alpha = \langle a, c\rangle \cdot \alpha'$ for some $c \in \mathbb{N}$, and $Val' = Val[x \leftarrow c]$. The symbol a is popped from the stack (if it is the top-most symbol), and its value is copied to the variable x.

We define the *initial configuration* $\gamma_{init} := \langle q_{init}, Val_{init}, \epsilon\rangle$, where $Val_{init}(x) = 0$ for all $x \in \mathcal{V}$. In other words, we start from a configuration where the automaton is in its initial state, the values of all variables are equal to 0, and the stack is empty (the fact that we choose to initialize the variables to 0 is not crucial for solving the problem).

For a configuration and a state $q \in Q$, we write $\gamma \xrightarrow{*} q$ to denote that $\gamma \xrightarrow{*} \gamma' = \langle q, Val, \alpha\rangle$ for some $Val : \mathcal{V} \mapsto \mathbb{N}$ and $\alpha \in (A \times \mathbb{N})^*$.

In other words, from γ we can reach a configuration whose state is q.

Reachability Problem. In the reachability problem PDAD-REACH, given a PDAD $\mathcal{A} = \langle Q, q_{init}, A, \Delta\rangle$ and a state $q_{target} \in Q$, we ask whether $\gamma_{init} \xrightarrow{*} q_{target}$.

4 Context-Free Grammars with Data

In this section, we introduce *Context-Free Grammars with Data* (CFGD) that are extensions of the classical model of Context-Free Grammars (CFG) in which (terminal and non terminal) symbols are defined by terms with free variables and productions have conditions defined by gap order constraints. We define the model, the operational semantics, and the reachability problem.

Model. A *Context-Free Grammars with Data* (CFGD) is a tuple $\mathcal{G} = \langle \mathcal{S}, X_{init}, P\rangle$, where $\mathcal{S}$ is a finite set of *symbols*. $X_{init} \in \mathcal{S}$ is the *start (or initial) symbol*, and P is the set of *productions*. Each symbol X has an *arity* $\rho(X) \in \mathbb{N}$ that is a natural number. Without loss of generality, we assume that $\rho(X_{init}) = 1$. A *term* has the form $X(x_1, \ldots, x_n)$ where $X \in \mathcal{S}$, $\rho(X) = n$ and $x_1, \ldots x_n \in \mathcal{V}$ are variables. A *ground term* has the form $X(c_1, \ldots, c_n)$ where $X \in \mathcal{S}$, $\rho(X) = n$ and $c_1, \ldots c_n \in \mathbb{N}$ are natural numbers. For a term σ of the form $X(x_1, \ldots, x_n)$ we define $Sym(\sigma) = X$ and $Var(\sigma) = \{x_1, \ldots, x_n\}$. We define $Sym(\sigma)$ for a ground term σ similarly. A *(ground) sentence* α is a finite set $\{\sigma_1, \sigma_2, \cdots, \sigma_n\}$, where each σ_i is a (ground) term. We define $Sym(\alpha) := \{Sym(\sigma_1), \ldots, Sym(\sigma_n)\}$, i.e., it is the set of symbols that occur in α. For a term $\sigma = X(x_1, \ldots, x_n)$ and a valuation Val, we define $Val(\sigma) := X(Val(x_1), \ldots, Val(x_n))$ to be the ground term we get by substituting each variable x_i in σ by $Val(x_i)$. For a sentence α, we define $Val(\alpha)$ similarly.

A *condition* θ is a finite conjunction of formulas of the forms: $x <_c y$ or $x = y$, where $x, y \in \mathcal{V}$ and $c \in \mathbb{N}$. Here $x <_c y$ stands for $x + c < y$. Sometimes, we treat a condition θ as set, and write e.g. $(x <_c y) \in \theta$ to indicate that $x <_c y$ is one of the conjuncts in θ. For a valuation Val, we use $Val(\theta)$ to denote the result of substituting each variable x in θ by $Val(x)$. We use $Val \models \theta$ to denote that $Val(\theta)$ evaluates to *true*. We use $Var(\theta)$ to denote the set of variables that occur in θ.

A *production* p is of the form $\sigma \rightsquigarrow \alpha \ : \ \theta$, where σ is a term, α is a non-empty sentence, and θ is a condition. We often use the notation $\sigma \rightsquigarrow \sigma_1 \cdots \sigma_n \ : \ \theta$ to denote the production $\sigma \rightsquigarrow \{\sigma_1, \ldots, \sigma_n\} \ : \ \theta$ (i.e. a sequence in the right-hand side denotes a set of terms). We use $\mathcal{N}$ to denote the set of non-terminals consisting of symbols that occur in the left-hand side of a production (we say that they are defined by a production). We use $\mathcal{T}$ to denote the set of terminals consisting of symbols that do not occur in the left-hand side of a production. Furthermore, we use $\mathcal{A}_T$ to denote the set of ground terms with symbols in $\mathcal{T}$.

Transition System. A *configuration* γ is a ground sentence. We define a transition relation $\longrightarrow_{\mathcal{G}}$ on the set of configurations by $\longrightarrow_{\mathcal{G}} := \cup_{p \in P} \xrightarrow{p}$ where $\xrightarrow{p}$ represents the effect of applying the production p. More precisely, for a production $p \in P$ of the form $\sigma \rightsquigarrow \alpha \ : \ \theta$, we have $\gamma_1 \xrightarrow{p} \gamma_2$ if there is a valuation $Val \models \theta$ such that $\gamma_1 = \alpha' \cup \{ Val(\sigma) \}$ and $\gamma_2 = \alpha' \cup \{ Val(\alpha) \}$.

For a set S of ground terms, we define $Pre(S)$ to be the set of ground terms σ which can, through the single application of a production, generate a configuration $\gamma \subseteq S$ (i.e., $\sigma \longrightarrow_{\mathcal{G}} \gamma$). Let $Pre^*(\cdot)$ denote the transitive closure of $Pre(\cdot)$.

We will use the following lemmata later in the paper.

Lemma 1. *Let α be a ground sentence of $\mathcal{G}$. Then, if for every ground term $\sigma \in \alpha$, we have $\sigma \xrightarrow{*}_{\mathcal{G}} \alpha''$ for some ground sentence α'' such that $Sym(\alpha'') \subseteq \mathcal{T}$, then $\alpha \xrightarrow{*}_{\mathcal{G}} \alpha'$ for α' such that $Sym(\alpha') \subseteq \mathcal{T}$.*

Lemma 2. *Let S be a set of ground terms and σ be a ground term such that $\sigma \in Pre^*(S)$. If $\sigma \notin S$ then there is a ground term $\sigma' \in (Pre(S) \setminus S)$.*

Reachability Problem. In the *reachability* problem CFGD-REACH, we are given a CFGD $\mathcal{G} = \langle \mathcal{S}, X_{init}, P \rangle$ and we are asked the question whether $X_{init}(0) \xrightarrow{*}_{\mathcal{G}} \alpha$ for some ground sentence α such that $Sym(\alpha) \subseteq \mathcal{T}$. In other words, we start from a configuration consisting of the start symbol with its parameter set to zero, and ask whether the system can reach a configuration where all its ground terms have symbols in $\mathcal{T}$.

CFGD *vs* CFG A Context-Free Grammars (CFG) is defined by production of the form $S \rightarrow w$ where w is a word defined over terminal and non terminal symbols. We can encode a CFG as a CFGD by associating to each terminal/non terminal symbol X (except the initial) a term $X(a, b)$ in which (a, b) are used to maintain an order in the right-hand side of a rule. For instance, the production $S \rightarrow SaS$ is encoded via the CFGD production $S(x, y) \rightarrow \{S(x, z), a(z, t), S(t, y)\} : x < z, z < t, t < y$.

CFGD *vs* CMRS CFGD also differ from the CMRS model [7]. CMRS is obtained by combining multiset rewriting and Gap Order constraints and it is aimed at modeling concurrent processes. CMRS rules have multiple heads and work over multisets of monadic terms (i.e. with a single argument, no nested terms). Differently from CMRS, CFGD productions have a single term in the left-hand side

and a set of terms in the right-hand side. This implies that multiple occurrences (with the same variables) of a term like $p(x, y)$ are counted only once. Furthermore, non-terminal symbols have arbitrary finite arity.

5 Symbolic Encoding

In this section, we define the symbolic representation used in the definition of the reachability algorithm (Section 6). The algorithm operates on *constraints*, where each constraint ϕ characterizes a (potentially) infinite set $[\![\phi]\!]$ of ground terms. A *constraint* ϕ is of the form $\sigma : \theta$ where σ is a term and θ is a condition. We define $Sym(\phi) = Sym(\sigma)$ and $Var(\phi) = Var(\sigma) \cup Var(\theta)$.

Definition 3. *The constraint ϕ characterizes a set of ground terms defined by $[\![\phi]\!] = \{\sigma' | \exists Val.\, (Val \models \theta) \wedge (\sigma' = Val(\sigma)\}$. For a finite set of constraints Φ, $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$.*

Without loss of generality, we can assume that $Var(\theta) = Var(\sigma)$, and that θ is consistent (constraints with inconsistent conditions characterize empty sets of configurations, and can therefore be safely discarded from the reachability analysis). A term $X(x_1, \ldots, x_n)$ is said to be *pure* if $x_i \neq x_j$ whenever $i \neq j$. A constraint $\sigma : \theta$ is said *pure* if σ is pure. We can assume without loss of generality that all constraints are pure. The reason is that if a variable x occurs (say) twice then the two occurrences of x can be replaced by two different variables y_1 and y_2 provided that we add a new conjunct $y_1 = y_2$ to the condition θ. For constraints ϕ_1, ϕ_2, we use $\phi_1 \sqsubseteq \phi_2$ to denote that ϕ_1 *subsumes* ϕ_2, i.e., $[\![\phi_1]\!] \supseteq [\![\phi_2]\!]$. Then, it is easy to see that checking whether $\phi_1 \sqsubseteq \phi_2$ can be reduced to the satisfiability problem for an existential Presburger formula (which is known to be NP-COMPLETE [34]).

Lemma 4. *For constraints ϕ_1, ϕ_2, the problem of checking whether $\phi_1 \sqsubseteq \phi_2$ is decidable.*

The following lemma states that we can transform any constraint ϕ of the form $\sigma : \theta$ to an equivalent constraint *clean*(ϕ) of the form $\sigma : \theta'$ such that $Var(\theta') = Var(\sigma)$ (i.e., we remove the extra-variables $(Var(\theta) \setminus Var(\sigma))$ from θ in order to satisfy the assumption that $Var(\theta) = Var(\sigma)$).

Lemma 5. *[31] Given a constraint ϕ of the form $\sigma : \theta$, we can construct a constraint clean(ϕ) of the form $\sigma : \theta'$ such that $Var(\theta') = Var(\sigma)$ and $[\![clean(\phi)]\!] = [\![\phi]\!]$.*

Given two terms σ_1 and σ_2, we say that σ_1 matches σ_2 iff $Sym(\sigma_1) = Sym(\sigma_2)$. For matching terms $\sigma_1 = X(x_1, \ldots, x_n)$ and $\sigma_2 = X(y_1, \ldots, y_n)$, where σ_2 is pure, we define $Ren_{\sigma_1}^{\sigma_2}$ to be a renaming such that $Ren_{\sigma_1}^{\sigma_2}(y_i) = x_i$ for all $i : 1 \leq i \leq n$. Consider a production $p = \sigma \rightsquigarrow \sigma_1 \cdots \sigma_n \ : \ \theta$ and constraints $\phi_1 = \sigma'_1 : \theta_1, \ldots, \phi_n = \sigma'_n : \theta_n$ such that σ_i and σ'_i are matching, and such that σ'_i is pure for all $i : 1 \leq i \leq n$. We define $p \otimes \phi_1 \otimes \cdots \otimes \phi_n$ to be the constraint

$\sigma : \theta \wedge Ren^{\sigma'_1}_{\sigma_1}(\theta_1) \wedge \cdots \wedge Ren^{\sigma'_n}_{\sigma_n}(\theta_n)$. For a set Φ of constraints, and production $p \in P$, we define $Pre_p(\Phi) := \{clean(\phi') \mid \exists \phi_1, \ldots, \phi_n \in \Phi. \phi' = p \otimes \phi_1 \cdots \otimes \phi_n\}$. We define $Pre(\Phi) := \cup_{p \in P} Pre_p(\Phi)$. Intuitively, $Pre(\Phi)$ defines a finite set of constraints that characterize the terms which can, through the single application of a production, generate a set of terms each of which belongs to Φ.

Lemma 6. $\bigcup_{\phi' \in Pre(\Phi)} [\![\phi']\!] = Pre([\![\Phi]\!])$.

For the set $\mathcal{T}$ of terminals, we define

$$\Phi_{\mathcal{T}} := \{a(x_1, \ldots, x_n) : \mathsf{true} \mid a \in \mathcal{T},\ \rho(a) = n\}$$

Notice that $\Phi_{\mathcal{T}}$ denotes the set of configurations whose symbols are in $\mathcal{T}$.

6 Reachability Analysis

In this section, we present an algorithm for solving the reachability analysis problem for CFGDs, and prove its partial correctness. The algorithm (Algorithm 1) inputs a CFGD $\mathcal{G} = \langle \mathcal{S}, X_{init}, P \rangle$ and answers the question whether we can reach a sentence where all the occurring terms are in $\mathcal{A}_{\mathcal{T}}$ (i.e. terms with symbols in $\mathcal{T}$). The algorithm maintains two sets of constraints: a set `ToExplore`, initialized to $\Phi_{\mathcal{T}}$, of constraints that have not yet been analyzed; and a set `Explored`, initialized to the empty set, of constraints that contain constraints that have already been analyzed.

The algorithm preserves the following four invariants:

1. For each $\sigma \in [\![\texttt{ToExplore} \cup \texttt{Explored}]\!]$, $\sigma \xrightarrow{*} \alpha$ for some α s.t. $Sym(\alpha) \subseteq \mathcal{T}$.
2. If $X_{init}(0) \xrightarrow{*} \alpha$ for some α s.t. $Sym(\alpha) \subseteq \mathcal{T}$, then there is a ground term $\sigma \in [\![\texttt{ToExplore}]\!]$ such that $\sigma \notin [\![\texttt{Explored}]\!]$.
3. $X_{init}(0) \notin [\![\texttt{Explored}]\!]$.
4. $[\![\Phi_{\mathcal{T}}]\!] \subseteq [\![\texttt{ToExplore} \cup \texttt{Explored}]\!]$.

Algorithm 1: Reachability analysis for a CFGD.

Input: A CFGD $\mathcal{G} = \langle \mathcal{S}, X_{init}, P \rangle$
Output: Is there a subset of terminal symbols $T \subseteq \mathcal{T}$ reachable in $\mathcal{G}$?

1 `ToExplore` $\leftarrow \Phi_{\mathcal{T}}$;
2 `Explored` $\leftarrow \emptyset$
3 **while** `ToExplore` $\neq \emptyset$ **do**
4   remove some ϕ from `ToExplore`;
5   **if** $X_{init}(0) \in [\![\phi]\!]$ **then return** *true*;
6   **else if** $\exists \phi' \in$ `Explored`. $\phi' \sqsubseteq \phi$ **then** discard ϕ;
7   **else**
8   `ToExplore` $\leftarrow$ `ToExplore` $\cup\ Pre(\texttt{Explored} \cup \{\phi\})$;
9   `Explored` $\leftarrow \{\phi\} \cup \{\phi' \mid \phi' \in \texttt{Explored} \wedge (\phi \not\sqsubseteq \phi')\}$;
10 **return** *false*

It is easy to see that the third and fourth invariants will be preserved. More precisely, for the third invariant, `Explored` is initially empty, and the condition at line 5 prevents adding any constraint whose symbol is X_{init} and parameter equals to 0 to `Explored`. The fourth invariant holds initially since $\texttt{ToExplore} \cup \texttt{Explored} = \Phi_{\mathcal{T}} \cup \emptyset = \Phi_{\mathcal{T}}$. This invariant is preserved since each time we remove a constraint from `ToExplore` (line 4), it is either eventually moved to `Explored` (line 9), or (in case it is discarded at line 6) there is already a constraint $\phi' \in \texttt{Explored}$ with $[\![\phi']\!] \supseteq [\![\phi]\!]$. Also, each time we remove a constraint ϕ' from `Explored` (line 9), we add the constraint ϕ to `Explored` where $[\![\phi]\!] \supseteq [\![\phi']\!]$.

Below, we show that the first two invariants are also preserved. Initially, the first invariant holds since $(\texttt{ToExplore} \cup \texttt{Explored}) = \Phi_{\mathcal{T}}$. The second invariant also holds initially since $\texttt{Explored} = \emptyset$ and $[\![\texttt{ToExplore}]\!] = [\![\Phi_{\mathcal{T}}]\!] \neq \emptyset$. Due to the first two invariants, the following two conditions can be checked during each step of the algorithm:

- From the second invariant, if `ToExplore` becomes empty then the algorithm terminates with a negative answer.
- From the first invariant, if a constraint ϕ is detected such that $X_{init}(0) \in [\![\phi]\!]$, then the algorithm terminates with a positive answer.

If neither of the two conditions is satisfied, the algorithm proceeds by picking and removing a constraint ϕ from `ToExplore`. Two possibilities arise depending on the value of σ:

- If there exists a constraint $\phi' \in \texttt{Explored}$ with $\phi' \sqsubseteq \phi$, then we discard ϕ. The first invariant is preserved since this operation will not add any new elements to $[\![\texttt{ToExplore} \cup \texttt{Explored}]\!]$. If $X_{init}(0) \xrightarrow{*} \alpha$ for some α s.t. $Sym(\alpha) \subseteq \mathcal{T}$, then the second invariant and the fact that $[\![\phi]\!] \subseteq [\![\texttt{Explored}]\!]$ imply that there is still some $\sigma \in \texttt{ToExplore}$ such that $\sigma \notin [\![\texttt{Explored}]\!]$. This means that the second invariant will also be preserved by this step.

- Otherwise, we compute the elements of $Pre(\texttt{Explored} \cup \phi)$, add them in `ToExplore`, move ϕ to `Explored`, and remove all constraints in `Explored` that are subsumed by ϕ. Let $\texttt{Explored}^{\texttt{old}}$ and $\texttt{Explored}^{\texttt{new}}$ be the contents of the set `Explored` before resp. after performing the operation. Define $\texttt{ToExplore}^{\texttt{old}}$ and $\texttt{ToExplore}^{\texttt{new}}$ analogously. The operation preserves the first invariant as follows. Pick any $\sigma \in [\![\texttt{ToExplore}^{\texttt{new}} \cup \texttt{Explored}^{\texttt{new}}]\!]$. If $\sigma \in [\![\texttt{ToExplore}^{\texttt{old}} \cup \texttt{Explored}^{\texttt{old}}]\!]$ then the result follows by the first invariant. Otherwise we know that $\sigma \in [\![Pre(\texttt{Explored}^{\texttt{old}} \cup \{\phi\})]\!]$, i.e., $\sigma \longrightarrow_{\mathcal{G}} \alpha$ where $\alpha \subseteq [\![\texttt{Explored}^{\texttt{old}} \cup \{\phi\}]\!]$ (see Lemma 6). By the induction hypothesis and the first invariant, we know that every ground term $\sigma' \in \alpha$, $\sigma' \xrightarrow{*}_{\mathcal{G}} \alpha'$ for some α' s.t. $Sym(\alpha') \subseteq \mathcal{T}$. Hence $\alpha \xrightarrow{*}_{\mathcal{G}} \alpha''$ for some α'' s.t. $Sym(\alpha'') \subseteq \mathcal{T}$ (see Lemma 1). In other words, $\sigma \longrightarrow_{\mathcal{G}} \alpha \xrightarrow{*}_{\mathcal{G}} \alpha''$ s.t. $Sym(\alpha'') \subseteq \mathcal{T}$. The operation also preserves the second invariant as follows. Assume that $X_{init}(0) \xrightarrow{*}_{\mathcal{G}} \alpha$ for some α s.t. $Sym(\alpha) \subseteq \mathcal{T}$. There are two cases. If there is a $\sigma \in [\![\Phi_{\mathcal{T}}]\!]$ such that $\sigma \notin [\![\texttt{Explored}^{\texttt{new}}]\!]$, then

by the fourth invariant $\sigma \in [\![\texttt{ToExplore}^{\texttt{new}}]\!]$ and the invariant holds immediately. Otherwise, $[\![\varPhi_{\mathcal{T}}]\!] \subseteq [\![\texttt{Explored}^{\texttt{new}}]\!]$. Since $X_{init}(0) \xrightarrow{*}_{\mathcal{G}} \alpha$ we have also that $X_{init}(0) \in \mathit{Pre}^* ([\![\texttt{Explored}^{\texttt{new}}]\!])$. By the third invariant, we know that $X_{init}(0) \notin [\![\texttt{Explored}^{\texttt{new}}]\!]$. By Lemma 2 that there is a ground term $\sigma \in (\mathit{Pre}([\![\texttt{Explored}^{\texttt{new}}]\!]) \setminus [\![\texttt{Explored}^{\texttt{new}}]\!])$. Since $[\![\texttt{Explored}^{\texttt{new}}]\!] = [\![\texttt{Explored}^{\texttt{old}} \cup \{\phi\}]\!]$ it follows that $\sigma \in [\![\mathit{Pre}(\texttt{Explored}^{\texttt{old}} \cup \{\phi\})]\!]$ and hence $\sigma \in [\![\texttt{ToExplore}^{\texttt{new}}]\!]$.

This give us the following theorem.

Theorem 7. *Algorithm 1, under termination assumption, always return the correct answer.*

7 Termination

In this section, we show that Algorithm 1 is guaranteed to terminate. To do that, we first recall some basics of the theory of well and better quasi-orderings. Then, we introduce a new class of constraints that we call *flat constraints* and show that they are better quasi-ordered. We show that each condition can be translated into a number of flat constraints. We use this to show that the set of conditions is well quasi-ordered under set inclusion. This leads to the well quasi-ordering of the set of constraints (of Section 5). Finally, we show the termination of the algorithm.

Wqo*s and* Bqo*s.* A *Quasi-Ordering* (or a Qo for short), is a pair $\langle A, \preceq\rangle$ where $\preceq$ is a reflexive and transitive binary relation on the set A. A QO $\langle A, \preceq\rangle$ is a *Well Quasi-Ordering* (Wqo), if for each infinite sequence $a_1, a_2, a_3, \ldots$ of elements of A , there are $i < j$ such that $a_i \preceq a_j$. The following lemma follows from the definition of a Wqo.

Lemma 8. *For* Qo*s $\preceq$ and $\preceq'$ on some set A, if $\preceq \subseteq \preceq'$ and $\preceq$ is a* Wqo *then $\preceq'$ is a* Wqo*.*

Given a Qo $\langle A, \preceq\rangle$, we define a Qo $\langle A^*, \preceq^*\rangle$ on the set of words A^* such that $a_1 a_2 \cdots a_m \preceq^* a'_1 a'_2 \cdots a'_n$ if there is an injection $h : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ such that $i < j$ implies $h(i) < h(j)$ for all $i, j : 1 \leq i, j \leq m$, and $a_i \preceq a'_{h(i)}$ for each $i : 1 \leq i \leq m$. We define the relation $\preceq^{\mathcal{P}}$ on the powerset $\mathcal{P}(A)$ (finite set of elements in A) of A, so that $A_1 \preceq^{\mathcal{P}} A_2$ if $\forall a_2 \in A_2 . \exists a_1 \in A_1 . a_1 \preceq a_2$.

We define the relation $\preceq^p$ on the Cartesian product $A_1 \times \ldots \times A_n$ of orders $\langle A_i, \leq_i\rangle$ for $i : 1, \ldots, n$, so that $\langle a_1, \ldots, a_n\rangle \preceq^p \langle a'_1, \ldots, a'_n\rangle$ if $a_i \preceq_i a'_i$ for $i : 1, \ldots, n$.

In the following lemma we state some properties of Bqos[1] [10,28].

[1] The technical definition of Bqos is quite complicated and can be found in e.g. [10]. The actual definition is not needed for understanding the rest of the paper, and is therefore omitted here.

Lemma 9. – *Each* BQO *is* WQO.
- *If A is finite, then $\langle A, = \rangle$ is a* BQO, *and $\langle \mathcal{P}(A), \subseteq \rangle$ is a* BQO.
- $\langle \mathbb{N}, \leq \rangle$ *is a* BQO.
- *If $\langle A_i, \leq_i \rangle$ is a* BQO *for $i : 1, \ldots, n$ then $\langle A_1 \times \ldots \times A_n, \preceq^p \rangle$ is a* BQO.
- *If $\langle A, \preceq \rangle$ is a* BQO, *then $\left\langle \mathcal{P}(A), \preceq^{\mathcal{P}} \right\rangle$ is a* BQO.

Flat Constraints. Fix a set $\mathcal{V} = \{x_1, \ldots, x_n\}$ of variables. A flat constraint ψ over $\mathcal{V}$ if of the form $A_0 c_1 A_1 \cdots c_m A_m$, where $c_1, \ldots, c_m \in \mathbb{N}$, and $A_0, A_2, \ldots, A_m$ is a partitioning of $\mathcal{V}$, i.e., $\mathcal{V} = A_0 \cup A_1 \cup \cdots \cup A_m$, $A_i \neq \emptyset$, and $A_i \cap A_j = \emptyset$ if $i \neq j$. In other words, a flat constraint is a word which alternatively contains sets of variables and natural numbers, starting and ending with a set of variables. The flat constraint ψ characterizes an infinite set $[\![\psi]\!]$ of vectors over $\mathbb{N}$ of length n, i.e., $[\![\psi]\!] \subseteq \mathbb{N}^n$. More precisely, define $h_\psi : \{1, \ldots, n\} \mapsto \{0, \ldots, m\}$ such that $h_\psi(i) = k$ if $x_i \in A_k$. $v = \langle d_1, \ldots, d_n \rangle \in [\![\psi]\!]$ iff the following conditions are satisfied for all $i, j : 1 \leq i, j \leq n$:

- $d_i = d_j$ if $h_\psi(i) = h_\psi(j)$.
- If $h_\psi(i) = k$. and $h_\psi(j) = k + 1$ then $c_{k+1} < d_j - d_i$.

In other words, the variable x_i represents d_i in ψ. If two variables are mapped to the same set then their values should be identical. Furthermore, the natural numbers c_i define the gaps between values of variables belonging to the different sets. For flat constraints $\psi = A_0 c_1 A_1 \cdots c_m A_m$ and $\psi' = A'_0 c'_1 A'_1 \cdots c'_m A'_m$ over $\mathcal{V}$, we write $\psi \preceq \psi'$ to denote that (i) $A'_i = A_i$ for all $i : 0 \leq i \leq m$, and (ii) $c_i \leq c'_i$ for all $i : 1 \leq i \leq m$. The following lemma follows from the definitions.

Lemma 10. *$\psi \preceq \psi'$ implies that $[\![\psi]\!] \supseteq [\![\psi']\!]$.*

By Lemma 9 it follows that

Lemma 11. *$\preceq$ is a* BQO *on the set of flat constraints.*

Proof. We first observe that flat contraints can be viewed as tuples with at most $K = |\mathcal{V}|$ partitions and $|\mathcal{V}| - 1$ constants and we can always add finite sequences such as $0\emptyset 0 \ldots 0\emptyset$ to consider K-tuples only. From Lemma 9, we know that $\langle \mathbb{N}, \leq \rangle$ and $\langle \mathcal{P}(\mathcal{V}), = \rangle$ are BQOs. Thus, the Cartesian product $(\mathcal{P}(\mathcal{V}) \times \mathbb{N})^{K-1} \times \mathcal{P}(\mathcal{V})$ with $\preceq$ is still a BQO.

Flattening. Consider a condition θ with $Var(\theta) = \{x_1, \ldots, x_n\}$ (recall the definitions of conditions and constraints from Section 5). We define $[\![\theta]\!]$ to be the set of vectors $v = \langle d_1, \ldots, d_n \rangle \in \mathbb{N}^n$, such that there is a valuation Val with $Val \models \theta$ and $Val(x_i) = d_i$ for all $i : 1 \leq i \leq n$. Furthermore, for two conditions on the same set of variables we define $\theta \sqsubseteq \theta'$ iff $[\![\theta]\!] \supseteq [\![\theta']\!]$. A *flattening* of θ is a flat constraint ψ over $Var(\theta)$, of the form $A_0 c_1 A_1 \cdots c_m A_m$ where $c_1, \ldots, c_m \geq 0$ are minimal natural numbers such that the following conditions are satisfied:

- If $(x = y) \in \theta$ then $x, y \in A_i$ for some $i : 1 \leq i \leq m$.
- If $(x <_c y) \in \theta$, $x \in A_i$, and $y \in A_j$ then $c \leq \left(\sum_{k=i+1}^{j} (c_k + 1) - 1 \right)$.

Intuitively, variables which are required to be equal by θ, are put in the same X_i. Also, variables which are ordered according to θ, are placed sufficiently far apart to cover the corresponding gap. We define $\mathcal{F}(\theta)$ to be the set of flattening of θ. In general conditions induce a partial order between variables. The flattening contains all linearizations with minimal gaps (constants) between variables. Notice that this set is finite. As an example, consider the condition $x <_2 y, x <_1 z$. Since there are no constraints on y and z, we have three different flattening where $y < z$ or $y = z$ or $y > z$, namely $\{x\}2\{y\}0\{z\}$, $\{x\}2\{y,z\}$, and $\{x\}1\{z\}0\{y\}$.

We define an ordering $\preceq$ on conditions such that $\theta \preceq \theta'$ if for each $\psi' \in \mathcal{F}(\theta')$ there is a $\psi \in \mathcal{F}(\theta)$ with $\psi \preceq \psi'$. From Lemma 10 we get the following.

Lemma 12. *$\theta \preceq \theta'$ implies that $[\![\theta]\!] \supseteq [\![\theta']\!]$.*

The following lemma follows from Lemma 9 and Lemma 11.

Lemma 13. *$\preceq$ is a* BQO *(and hence* WQO*) on the set of conditions.*

From Lemma 13, Lemma 12, and Lemma 8 we get the following lemma.

Lemma 14. *The set of conditions is* WQO *under $\sqsubseteq$.*

The following lemma then holds.

Lemma 15. *The set of constraints is* WQO *under $\sqsubseteq$.*

Proof. Consider an infinite sequence of constraints: $\phi_1, \phi_2, \phi_3, \ldots$. Since the set $\mathcal{N} \cup \mathcal{T}$ is finite, there is an infinite sequence $i_1 < i_2 < i_3 < \cdots$ such that $Sym(\phi_{i_1}) = Sym(\phi_{i_2}) = Sym(\phi_{i_3}) = \cdots$. If $Sym(\phi_{i_j}) \in \mathcal{T}$ then the result follows immediately (since $[\![\phi_{i_j}]\!] = \{Sym(\phi_{i_j})\}$ for all $j \geq 1$). Otherwise, we can assume, without loss of generality, that ϕ_{i_j} is of the form $X(x_1, \ldots, x_n) : \theta_{i_j}$. Notice that each $Var(\theta_{i_j}) = \{x_1, \ldots, x_n\}$ is a condition over $\{x_1, \ldots, x_n\}$. By Lemma 14, there are $j < k$ such that $\theta_{i_j} \sqsubseteq \theta_{i_k}$, and hence $\phi_{i_j} \sqsubseteq \phi_{i_k}$.

Termination. The reason why the algorithm always terminates is that only a finite set of constraints can be added to `Explored`. This can be explained as follows. By definition, a new element ϕ is added to `Explored` only if $\phi' \not\sqsubseteq \phi$, for each ϕ' already added to `Explored`. This means that the constraints added to `Explored` form a sequence $\phi_1, \phi_2, \phi_3, \ldots$, such that $\phi_i \not\sqsubseteq \phi_j$ for all $i < j$. By WQO of $\sqsubseteq$ (Lemma 15) it follows that this sequence is finite. This gives the following theorem.

Theorem 16. *Algorithm 1 is guaranteed to terminate.*

8 Translation

Reachability with Empty Stacks. We consider a different variant of PDAD-REACH which we call PDAD-REACH-EMPTY. An instance of PDAD-REACH-EMPTY is defined by a PDAD $\mathcal{A} = \langle Q, q_{init}, A, \Delta \rangle$ and a state $q_{target} \in Q$, and we are

asked whether $\gamma_{init} \xrightarrow{*} \gamma$ for some γ of the form $\langle q_{target}, Val, \epsilon \rangle$, i.e., we ask whether we reach q_{target} at a configuration where the stack is *empty*. Given an instance of PDAD-REACH, defined by a PDAD $\mathcal{A} = \langle Q, q_{init}, A, \Delta \rangle$ and a state $q_{target} \in Q$, we derive an equivalent instance of PDAD-REACH-EMPTY as follows. We construct a new PDAD $\mathcal{A}'$ from $\mathcal{A}$ by adding a new state q_{new} to Q, and adding a transition labeled with *nop* from q_{target} to q_{new}. For each member $a \in A$ of the stack alphabet, we add a self-loop on q_{new} that pops a (with any value). The two problem instances are equivalent as follows. Suppose that q_{new} is reachable with an empty stack in $\mathcal{A}'$. Then, the run of $\mathcal{A}'$ reaching q_{new} must have passed through q_{target} (since q_{new} can only be reached from q_{target}). This means that q_{target} is reachable in $\mathcal{A}$. On the other hand, suppose that q_{target} is reachable in $\mathcal{A}$. Then, $\mathcal{A}'$ can simulate the run of $\mathcal{A}$ until it reaches q_{target}. From there, it takes the transition to q_{new}, and starts executing the self-loops, popping all the symbols in the stack until the stack becomes empty.

From PDAD *to* CFGD. Suppose that we are given an instance of PDAP-REACH-EMPTY defined by a PDAD $\mathcal{A} = \langle Q, q_{init}, A, \Delta \rangle$ and a state $q_{target} \in Q$. Let $\{x_1, \ldots, x_n\}$ be the set of variables that occur in $\mathcal{A}$. We derive an equivalent instance of CFGD-REACH defined by a CFGD $\mathcal{G} = \langle \mathcal{S}, X_{init}, P \rangle$. The set $\mathcal{T}$ of $\mathcal{G}$ is defined by the singleton set $\{t\}$ and we assume that the arity of t is 0 (i.e., $\rho(t) = 0$). The set of $\mathcal{N}$ of $\mathcal{G}$ is defined as follows: For each pair of states $q_1, q_2 \in Q$ and symbol $a \in A \cup \{\bot\}$, with $\bot \notin A$, we have a nonterminal $X_{(q_1,a,q_2)} \in \mathcal{N}$ with arity $2n+1$. The symbol $\bot$ is used to denote that the stack of $\mathcal{A}$ is empty. The set of non-terminal set $\mathcal{N}$ contains the initial symbol X_{init} (by definition).

In the following, let $\bar{y}$ denote a vector $\langle y_1, \ldots, y_n \rangle$ of length n, and define $\bar{y}[i] := y_i$ for $i : 1 \leq i \leq n$. For vectors $\bar{z} = \langle z_1, \ldots, z_n \rangle$ and $\bar{y} = \langle y_1, \ldots, y_n \rangle$, we use $\bar{z} = \bar{y}$ (resp. $\bar{z} \neq_j \bar{y}$ for some $j : 1 \leq j \leq n$) to denote the condition $\bigwedge_{1 \leq i \leq n} z_i = y_i$ (resp. $\bigwedge_{(1 \leq i \leq n) \wedge (i \neq j)} z_i = y_i$). Furthermore, for brevity, we sometimes shorten a conjunction of conditions $\theta_1 \wedge \ldots \wedge \theta_n$ into a list $\theta_1, \ldots, \theta_n$.

Intuitively, a non-terminal of the form $X_{(q_1,a,q_3)}(\bar{y}, \bar{z}, \ell)$ represents a run of $\mathcal{A}$ from a configuration where the state is q_1, the topmost stack symbol is a and its corresponding value is given by the value ℓ (if $a = \bot$ then the stack is empty), and the valuation of the shared variables of $\mathcal{A}$ is given by the valuation of $\bar{y}$, to a configuration with a stack content where a has been popped and where the state is q_3 and the valuation of the shared variables of $\mathcal{A}$ is given by the valuation of $\bar{z}$.

The set P is derived from Δ, and it contains the productions of Fig. 1. Then the following property holds.

Proposition 17. *$\gamma_{init} \xrightarrow{*} \gamma$ for some $\gamma = \langle q_{target}, Val, \epsilon \rangle$ iff $X_{init} \xrightarrow{*}_{\mathcal{G}} \alpha$ for some sentence α such that $Sym(\alpha) \subseteq \mathcal{T}$.*

As an immediate consequence of the above Proposition, Theorem 7, and Theorem 16, we get:

Theorem 18. *The* PDAD-REACH *and* PDAD-REACH-EMPTY *problems are decidable for* PDADs*.*

$$\frac{\langle q_1, nop, q_2\rangle \in \Delta \quad q_3 \in Q}{(X_{(q_1,a,q_3)}(\overline{y},\overline{z},\ell) \rightsquigarrow X_{(q_2,a,q_3)}(\overline{y'},\overline{z'},\ell') \ : \ \overline{y}=\overline{y'},\overline{z}=\overline{z'},\ell=\ell') \in P}$$

$$\frac{\langle q_1, x_i \leftarrow *, q_2\rangle \in \Delta \quad q_3 \in Q}{(X_{(q_1,a,q_3)}(\overline{y},\overline{z},\ell) \rightsquigarrow X_{(q_2,a,q_3)}(\overline{y'},\overline{z'},\ell') \ : \ \overline{y}\neq_i\overline{y'},\overline{z}=\overline{z'},\ell=\ell') \in P}$$

$$\frac{\langle q_1, x_i \leftarrow x_j, q_2\rangle \in \Delta \quad q_3 \in Q}{(X_{(q_1,a,q_3)}(\overline{y},\overline{z},\ell) \rightsquigarrow X_{(q_2,a,q_3)}(\overline{y'},\overline{z'},\ell') \ : \ \overline{y}\neq_i\overline{y'},\overline{z}=\overline{z'},\ell=\ell',\overline{y'}[i]=\overline{y}[j]) \in P}$$

$$\frac{\langle q_1, x_i \leftarrow (>_c x_j), q_2\rangle \in \Delta \quad q_3 \in Q}{(X_{(q_1,a,q_3)}(\overline{y},\overline{z},\ell) \rightsquigarrow X_{(q_2,a,q_3)}(\overline{y'},\overline{z'},\ell') \ : \ \overline{y}\neq_i\overline{y'},\overline{z}=\overline{z'},\ell=\ell',\overline{y}[j]<_c\overline{y'}[i]) \in P}$$

$$\frac{\langle q_1, x_j = x_i, q_2\rangle \in \Delta \quad q_3 \in Q}{(X_{(q_1,a,q_3)}(\overline{y},\overline{z},\ell) \rightsquigarrow X_{(q_2,a,q_3)}(\overline{y'},\overline{z'},\ell') \ : \ \overline{y}=\overline{y'},\overline{z}=\overline{z'},\ell=\ell',\overline{y}[i]=\overline{y}[j]) \in P}$$

$$\frac{\langle q_1, x_j <_c x_i, q_2\rangle \in \Delta \quad q_3 \in Q}{(X_{(q_1,a,q_3)}(\overline{y},\overline{z},\ell) \rightsquigarrow X_{(q_2,a,q_3)}(\overline{y'},\overline{z'},\ell') \ : \ \overline{y}=\overline{y'},\overline{z}=\overline{z'},\ell=\ell',\overline{y'}[j]<_c\overline{y}[i]) \in P}$$

$$\frac{\langle q_1, push\,(b)\,(x_i), q_2\rangle \in \Delta \quad q_3,q_4 \in Q}{(X_{(q_1,a,q_3)}(\overline{y},\overline{z},\ell) \rightsquigarrow X_{(q_2,b,q_4)}(\overline{y'},\overline{u},\ell')\,X_{(q_4,a,q_3)}(\overline{u'},\overline{z'},\ell'') \ : \ \overline{y}=\overline{y'},\overline{u}=\overline{u'},\overline{z}=\overline{z'},\ell=\ell'',\ell'=\overline{y}[i]) \in P}$$

$$\frac{\langle q_1, pop\,(x_i)\,(a), q_2\rangle \in \Delta}{(X_{(q_1,a,q_2)}(\overline{y},\overline{z},\ell) \rightsquigarrow t \ : \ \overline{y}\neq_i\overline{z},\overline{z}[i]=\ell) \in P}$$

$$\frac{}{(X_{init}(x) \rightsquigarrow X_{(q_{init},\perp,q_{target})}(\overline{y},\overline{z},\ell) \ : \ \bigwedge_{1\leqslant i\leqslant n}\overline{y}[i]=x) \in P}$$

$$\frac{}{(X_{(q_{target},\perp,q_{target})}(\overline{y},\overline{z},\ell) \rightsquigarrow t \ : \ \overline{y}=\overline{z}) \in P}$$

Fig. 1. From transitions of pushdown with data to productions

9 Extended PDADs

In this section, we present generalizations of the basic PDAD model for which the results presented in this paper still hold.

The first extension consists in adding to conditions of the form $x = c$, $x > c$, and $x < c$ for a variable x and a constant value $c \geq 0$. The resulting formulas corresponds to the original Gap Order Constraints considered in [31].

The second extension consists in adding multiple data fields in each element pushed to the stack. For fixed number of data fields $k \geq 0$, the configuration of PDAD$_k$ becomes a triple $\langle q, Val, \alpha\rangle$ where $q \in Q$ is a state, $Val : \mathcal{V} \mapsto \mathbb{N}$ is a valuation, and $\alpha \in \left(A \times \mathbb{N}^k\right)^*$ defines the content of the stack (each element of the word is a pair $\langle a, c_1, \ldots, c_k\rangle$ where a is the symbol and c_i is its value for the i-th field).

We now consider operations that manipulate the data fields. We first extend the push operation and consider $push\,(a)\,(x_1, \ldots, x_k)$ to push the symbol $a \in A$ and to assign to the i-th field the value of x_i for $i : 1, \ldots, k$. We also consider operation $pop\,(a)\,(x_1, \ldots, x_k)$ to pop the symbol $a \in A$ from the stack and to assign to x_i the value of the i-th field on the top of the stack $i : 1, \ldots, k$. The operational semantics can be naturally extended in order to cope with tuples of values instead of single one.

Finally, we consider operations that test and modify the data fields on the stack. We can use special identifiers $topx_1, \ldots, topx_k$ to denote such data fields and use them in conditions of transitions.

To encode the resulting model into CFGD, we need to introduce non-terminals with extra arguments that represent both the current value and the (guessed) updated value of data fields. More specifically, we need non-terminals of the form $X_{(q_1,a,q_2)}(\bar{x}, \bar{y}, \bar{z}, \bar{u})$ to represent a run of a $\mathcal{A}_k$ from a configuration where the state is q_1, the topmost stack symbol is a and its corresponding data field values are given by the vector $\bar{z}$, and the valuation of the shared variables of $\mathcal{A}$ is given by the valuation of $\bar{x}$, to a configuration with the updated data fields $\bar{u}$ and where the state is q_2 and the valuation of the shared variables is given by the valuation of $\bar{y}$.

We leave a detailed treatment of this extension for future work.

10 Related Work and Conclusion

Decidability and complexity of reachability problems for pushdown systems with or without data have been extensively studied in the literature. In [12] the authors present an algorithm to compute $Post^*$ and Pre^* for a pushdown automata and a regular set of its configurations (represented as automata). Symbolic versions of the algorithms have been studied e.g. in [29]. In [11] the authors consider approximated verification methods for subclasses of pushdown systems called finite indices in which it is possible to handle counters without zero test (i.e. transitions of a Petri net). In [2,1] the authors present decidability results for timed extensions of pushdown systems. In [14] the authors present decidability results for pushdown systems with either a well-quasi ordered set of control locations or of data values. In our model we do not consider a well-quasi ordered data domain, but introduce a well-quasi ordered relation over values pushed to and popped from the stack in order to decide reachability. Our extensions of pushdown system with Gap Order is orthogonal to the above mentioned models. Furthermore, it subsumes the model presented in [32], where the authors consider pushdown systems in which messages carry (object) identifiers that can be compared by equality. In addition to equality tests, Gap Order can be used to order messages in the stack.

Concerning our proof techniques, the algorithm for solving the CFGD reachability problem is inspired to the seminal results on Datalog and context-free language reachability [35,30] and to the evaluation of Datalog with Gap Order Constraints [31]. CLP programs with Gap Order constraints without conjunctions in the body have been used to model transition systems in [27]. The fixpoint semantics of CLP programs has been used to characterize model checking problems in [21] and applied to infinite-state systems in [18,16,17,20]. In [15] extended automata with Gap Order conditions over variables are used as an approximated model of counter systems. The model however does not have recursion. The complexity of verification problems (expressed in temporal logic) for transitions systems with Gap Order Constraints has been studied in [13]. Allowing rules with sets of terms in the right-hand side, GFGD are more general than

the model in [13]. Multiset rewriting systems with Gap Order Constraints (i.e. systems with an arbitrary number of integral variables) have been introduced in [3] and applied to different types of systems in [8] extending the parameterized models described in [9,22]. These systems are a subclass of multiset rewriting with (linear) constraints applied to infinite state verification, e.g., in [19].

The evaluation procedure for Datalog with Gap Order Constraints in [31] and its termination depend on specific data structures (weighted graphs kept in normal form) used to represent relations between variables that occur in Datalog clauses. In the present paper we formulate an algorithmic solution to CFGD reachability as an instance of the general framework of well-structured transition systems and apply the theory of better-quasi ordering to naturally infer its termination. This approach has the great advantage of capturing the essential ingredients needed for extending the algorithm to other classes of grammars with data. For instance, under some restrictions on the arity of terms, a slightly modified algorithm can be applied to grammars with sets of terms in the left-hand side of a production. A more formal treatment of this kind of generalization together with a deeper investigation of the complexity of the resulting algorithm is part of our future work.

References

1. Abdulla, P.A., Atig, M.F., Stenman, J.: Dense-timed pushdown automata. In: LICS. IEEE Computer Society (2012)
2. Abdulla, P.A., Atig, M.F., Stenman, J.: The minimal Cost reachability problem in priced timed pushdown systems. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 58–69. Springer, Heidelberg (2012)
3. Abdulla, P.A., Delzanno, G.: On the coverability problem for constrained multiset rewriting. In: Proc. AVIS 2006, 5th Int. Workshop on on Automated Verification of Infinite-State Systems (2006)
4. Abdulla, P.A.: Well (and Better) Quasi-Ordered Transition Systems. The Bulletin of Symbolic Logic 16(4), 457–515 (2010)
5. Abdulla, P.A., Atig, M.F., Cederberg, J.: Timed lossy channel systems. In: Proc. FSTTCS 2005, 32nd Conf. on Foundations of Software Technology and, Theoretical Computer Science (2012)
6. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Proc. LICS 1996, 11th IEEE Int. Symp. on Logic in Computer Science, pp. 313–321 (1996)
7. Abdulla, P.A., Delzanno, G., Begin, L.V.: A classification of the expressive power of well-structured transition systems. Inf. Comput. 209(3), 248–279 (2011)
8. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
9. Abdulla, P.A., Delzanno, G., Rezine, A.: Approximated parameterized verification of infinite-state processes with global conditions. Formal Methods in System Design 34(2), 126–156 (2009)
10. Abdulla, P.A., Nylén, A.: Better is better than well: On efficient verification of infinite-state systems. In: Proc. LICS 2000, 16th IEEE Int. Symp. on Logic in Computer Science, pp. 132–140 (2000)

11. Atig, M.F., Ganty, P.: Approximating Petri net reachability along context-free traces. In: FSTTCS 2011. LIPIcs, vol. 13, pp. 152–163. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
12. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
13. Bozzelli, L., Pinchinat, S.: Verification of gap-order constraint abstractions of counter systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 88–103. Springer, Heidelberg (2012)
14. Cai, X., Ogawa, M.: Well-structured extensions of pushdown systems. In: RP 2012 (2012)
15. Čerāns, K.: Deciding properties of integral relational automata. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 35–46. Springer, Heidelberg (1994)
16. Delzanno, G.: Automatic verification of parameterized cache coherence protocols. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)
17. Delzanno, G., Bultan, T.: Constraint-based verification of client-server protocols. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 286–301. Springer, Heidelberg (2001)
18. Delzanno, G., Esparza, J., Podelski, A.: Constraint-based analysis of broadcast protocols. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 50–66. Springer, Heidelberg (1999)
19. Delzanno, G., Ganty, P.: Automatic verification of time sensitive cryptographic protocols. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 342–356. Springer, Heidelberg (2004)
20. Delzanno, G.: Constraint-based verification of parameterized cache coherence protocols. Formal Methods in System Design 23(3), 257–301 (2003)
21. Delzanno, G., Podelski, A.: Model checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999)
22. Delzanno, G., Rezine, A.: A lightweight regular model checking approach for parameterized systems. STTT 14(2), 207–222 (2012)
23. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
24. Esparza, J., Kučera, A., Mayr, R.: Model checking probabilistic pushdown automata. In: Proc. LICS 2004, 20th IEEE Int. Symp. on Logic in Computer Science, pp. 12–21 (2004)
25. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
26. Etessami, K., Yannakakis, M.: Algorithmic verification of recursive probabilistic state machines. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 253–270. Springer, Heidelberg (2005)
27. Fribourg, L., Richardson, J.: Symbolic verification with gap-order constraints. In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 20–37. Springer, Heidelberg (1997)
28. Marcone, A.: Foundations of BQO theory. Transactions of the American Mathematical Society 345(2) (1994)

29. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
30. Reps, T.: Program analysis via graph reachability. Information & Software Technology 40(11–12), 701–726 (1998)
31. Revesz, P.Z.: A closed-form evaluation for datalog queries with integer (gap)-order constraints. TCS 116(1&2), 117–149 (1993)
32. Rot, J., de Boer, F.S., Bonsangue, M.M.: Pushdown System Representation For Unbounded Object Creation. Tech. Rep. KIT-13, Karlsruhe Institute of Technology (July 2010)
33. Schwoon, S.: Model-Checking Pushdown Systems. Ph.D. thesis, Technische Universität München (2002)
34. Verma, K.N., Seidl, H., Schwentick, T.: On the complexity of equational Horn clauses. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 337–352. Springer, Heidelberg (2005)
35. Yannakakis, M.: Graph-theoretic methods in database theory. In: PODS 1990, pp. 230–242 (1990)

Model Checking MANETs with Arbitrary Mobility

Fatemeh Ghassemi[1], Saeide Ahmadi[1], Wan Fokkink[2], and Ali Movaghar[3]

[1] University of Tehran, Tehran, Iran
[2] VU University Amsterdam, Amsterdam, The Netherlands
[3] Sharif University of Technology, Tehran, Iran

Abstract. Modeling arbitrary connectivity changes of mobile ad hoc networks (MANETs) makes application of automated formal verification challenging. We introduced constrained labeled transition systems (CLTSs) as a semantic model to represent mobility. To model check MANET protocol with respect to the underlying topology and connectivity changes, we here introduce a branching-time temporal logic interpreted over CLTSs. The temporal operators, from Action Computation Tree Logic with an unless operator, are parameterized by multi-hop constraints over topologies, to express conditions on successful scenarios of a MANET protocol. We moreover provide a bisimilarity relation with the same distinguishing power for CLTSs as our logical framework.

1 Introduction

In mobile ad hoc networks (MANETs), nodes communicate along multi-hop paths using wireless transceivers. Wireless communication is restricted; only nodes located in the range of a transmitter receive data. Due to e.g. noise in the environment, interferences, and temporary communication link errors, wireless communication is unreliable, which together with mobility of nodes complicates the design of MANET protocols. Formal methods provide valuable tools to design, evaluate and verify such protocols.

We introduced *Restricted Broadcast Process Theory* (*RBPT*) [9] to specify and verify MANETs, taking into account mobility. *RBPT* specifies a MANET by composing nodes using a restricted local broadcast operator. A strong point of *RBPT* is that the underlying topology is not specified in the syntax, which would make it hard to set up the initial topology for each scenario in a verification. In similar approaches, the mobility is modeled as arbitrary manipulation of the underlying topology (given as part of the semantic state), which may make the model infinite and insusceptible to automated verification techniques. Instead in the semantic model of *RBPT*, a *constraint labeled transition system* (*CLTS*) [10], transitions are enriched with so-called network constraints, to restrict the possible topologies. This symbolic representation of network topologies in the semantics is more compact and allows automated verification techniques to investigate families of properties on a unified model.

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 217–232, 2013.
DOI: 10.1007/978-3-642-40213-5_14,
© IFIP International Federation for Information Processing 2013

Properties in MANETs tend to be weaker than in wired networks, due to the topology-dependent behavior of communication, and consequently the need for multi-hop communication between nodes. For instance, an important property in routing or information dissemination protocols is *packet delivery*: If there exists an end-to-end route between two nodes A and B for a long enough period of time, then packets sent by A will be received by B [7]. To reason about properties that require such topology conditions, we introduce a temporal logic CACTL based on ACTLW [16], which consists of Action CTL [3] with an until operator. Our approach supports flexibility in verifying topology-dependent behavior (without changing the model), and restricting the generality of mobility as opposed to existing approaches. CACTL is interpreted over CLTSs. Path operators are parameterized with multi-hop constraints over the underlying topologies. We present a model checking algorithm for CACTL; a model checker for CACTL is being implemented, using the rewrite logic Maude. This provides a framework, supporting both equational reasoning [9] and model checking of MANET protocols, to verify topology-dependent properties like "existence of a route".

We moreover introduce a novel notion of branching network bisimilarity, based on branching bisimilarity [24], that induces the same identification of CLTSs as CACTL. This relation is finer than the one introduced in [10], due to reliability of communication: A receiving node is not equivalent to a deadlocked node anymore, since in parallel with a sending node, an unsuccessful communication cannot be matched to a communication with no enabled receiver (which is the case in the lossy framework).

2 Related Work

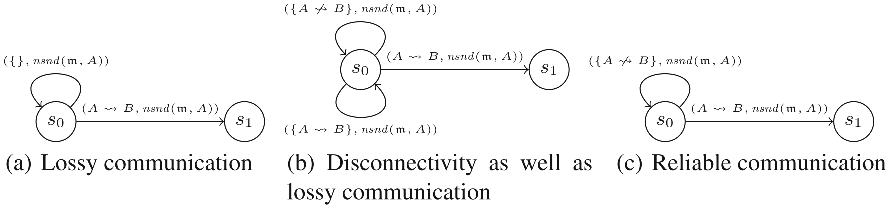
MANET protocols have been studied either using existing formalisms such as SPIN [1,5,26] and UPPAAL [8,15,26,27], or introducing specific frameworks mainly with an algebraic approach [7,13,14,17,18,20,21,23]. Important modeling challenges in MANETs are *local broadcast*, *underlying topology* and *mobility*. The modeling approach using existing formalisms can be summarized as follows: The underlying topology is modeled by a two-dimensional array of Booleans, mobility by explicit manipulation of this matrix, and local broadcast by unicasting to all nodes with whom the sending node is presently connected, using the connectivity matrix. The verification approach tends to be based on model checking techniques restricted to a pre-specified mobility scenario. Lack of support for compositional modeling and arbitrary topology changes has motivated new approaches with a primitive for local broadcast and support of arbitrary mobility. These approaches are CBS#, bKlaim, CWS, CMAN, CMN, ω-calculus, *RBPT*, CSDT, and AWN [7,9,13,14,17,18,20,21,23]. The common point among them (except *RBPT*) is implicit manipulation of the underlying topology in the semantics to model arbitrary connectivity changes and mobility. The analysis techniques supported by these frameworks, except bKlaim and AWN, are based on a behavioral congruence relation. In [10] we provided an axiomatization to derive that a specification of a MANET protocol is observably equal to a specification of its desired external behavior. Equational reasoning (applied at the

syntactic or the semantic level) requires either abstraction from the actual specification of the MANET protocol, or knowledge about the overall behavior of the MANET beforehand. The model checking approach is useful to investigate specific properties of MANET protocols with less effort and knowledge. The mix of broadcast behavior and mobility leads to state-space explosion, hampering the application of automated verification techniques like model checking. In bKlaim [21], the semantic model is abstracted to a finite labeled transition system such that the mobility information is preserved; a variant of ACTL is introduced to determine which properties hold if movement of nodes is restricted. To this aim, ACTL operators are parameterized by a set of possible network configurations (topology). However, topology-dependent behavior cannot be checked. AWN [7] verifies topology-dependent behavior properties using CTL [2], by treating a transition label carrying (dis)connectivity information as a predicate of its succeeding state [7] and defining predicates over the topology as part of the syntax. This approach can be extended to algebras, e.g. CMAN and ω-calculus, with (dis)connectivity information on transition labels. However, this approach needs auxiliary strategies to extract predicates from the states and transitions, to restrict connectivity changes during model checking and thus limit the state space. These challenges are tackled with the help of the model checker UPPAAL, by transforming AWN specifications to automata and exploiting an auxiliary automaton which statically restricts connectivity changes [8], similar to [15].

3 Background

Communication in wireless networks tends to be based on local broadcast: Only nodes that are located in the transmission area of a sender can receive. A node B is *directly connected to* a node A, if B is located within the transmission range of A. This asymmetric connectivity relation between nodes introduces a *topology* concept. A topology is a function $\gamma : Loc \rightarrow I\!P(Loc)$ where Loc denotes a finite set of (hardware) addresses A, B, C. We extend Loc with the unknown address ? to model open communications, which is helpful in giving semantics to MANETs in a compositional way.

Constrained labeled transition systems (CLTSs) [10] provide a semantic model for the operational behavior of MANETs. A transition label is a pair of an action and a *network constraint*, restricting the range of possible underlying topologies. A network constraint $\mathcal{C}$ is a set of connectivity pairs $\rightsquigarrow: Loc \times Loc$, where only the first address can be ?. In this setting, non-existence of connectivity information between two addresses in a network constraint can imply three consequences; we do not have any information about the link (this is helpful when the link has no effect on the evolution of a network), the link was disconnected, or the link exists, but due to unreliable communication, the communication was unsuccessful. To distinguish these cases from each other, we extend the network constraints of CLTSs with a set of disconnectivity pairs $\not\rightsquigarrow: Loc \times Loc$; while $B \rightsquigarrow A$ denotes that A is connected to B directly and consequently A can receive data sent by B, $B \not\rightsquigarrow A$ denotes that A is not connected to B



(a) Lossy communication (b) Disconnectivity as well as lossy communication (c) Reliable communication

Fig. 1. Modeling different communication behaviors: s_0 represents a state in which A broadcasts its data, while s_1 represents a state in which data has been transferred from A to B

directly and consequently cannot receive any message from B. In this setting, non-existence of connectivity information between two addresses in a network constraint means a lack of information. We write $\{B \rightsquigarrow A, C \ - \ B \not\rightsquigarrow D, E\}$ instead of $\{B \rightsquigarrow A, B \rightsquigarrow C, B \not\rightsquigarrow D, B \not\rightsquigarrow E\}$.

A network constraint $\mathcal{C}$ is said to be *well-formed* if $\forall \ell \rightsquigarrow \ell' \in \mathcal{C}\, (\ell' \neq ? \wedge \ell \not\rightsquigarrow \ell' \notin \mathcal{C})$ and $\forall \ell \not\rightsquigarrow \ell' \in \mathcal{C}\, (\ell' \neq ? \wedge \ell \rightsquigarrow \ell' \notin \mathcal{C})$. Let $\mathbb{C}$ denote the set of well-formed network constraints that can be defined over network addresses in *Loc*. Each network constraint $\mathcal{C}$ represents the set of network topologies that satisfy the (dis)connectivity pairs in $\mathcal{C}$, i.e., $\{\gamma \mid \mathcal{C} \subseteq \mathcal{C}_\Gamma(\gamma)\}$, where $\mathcal{C}_\Gamma(\gamma)$ extracts all one-hop (dis)connectivity information from γ. So the empty network constraint $\{\}$ denotes all possible topologies over *Loc*. Let Act_τ be the set of actions (including the silent action τ), ranged over by η.

A *CLTS* is of the form by $\langle S, \Lambda, \rightarrow, s_0\rangle$, with S a set of states, $\Lambda \subseteq \mathbb{C} \times Act_\tau$, $\rightarrow \subseteq S \times \Lambda \times S$ a transition relation, and $s_0 \in S$ the initial state. A transition $(s, (\mathcal{C}, \eta), s') \in \rightarrow$, denoted by $s \xrightarrow{(\mathcal{C},\eta)} s'$, expresses that a MANET protocol in state s with an underlying topology $\gamma \in \mathcal{C}$ can perform action η to evolve to state s'. Extending network constraints with disconnectivity pairs enables us to define different behaviors for the communication primitive (see Fig. 1). Furthermore, it allows us to reason about existence of a communication path between nodes, as will be explained in Section 4.1.

4 Constrained Action Computation Tree Logic

Properties of MANETs tend to be weaker than of wired networks, due to topology-dependent behavior of communication, and consequently the requirement of existence of a multi-hop communication path between nodes. CLTSs provide a suitable platform to verify topology-dependent properties, using the (dis)connectivity information encoded into the transition labels: While transitions are traversed to investigate a behavioral property, (dis)connectivity information is collected to verify the topology conditions on which the behavior depends. To this aim, we introduce a temporal logic based on Action CTL (ACTL) [3] which includes the *until* and *next* operators from CTL [2], parameterized with a set of actions. Recently a more expressive variant of ACTL

called ACTLW [16] was introduced, in which the *next* is replaced by an *unless* operator.

4.1 Concepts

Since the behavior of MANET protocols depends on the underlying topology of the network, many properties depend on constraints on this topology. For example, to examine whether a routing protocol can find a route from node A to node B, the existence of a multi-hop path from A to B is a pre-condition. Viewing a network topology as a directed graph, the simplest form of constraint consists of the (non-)existence of multi-hop relations between nodes.

As explained in Section 3, states in a CLTS do not hold information about the underlying topology. E.g., from the transition sequence $t_0 \xrightarrow{(\{A \rightsquigarrow B\}, \eta_1)} t_1 \xrightarrow{(\{B \rightsquigarrow C\}, \eta_2)} t_2$ we can infer that at the moment we reach t_1, B was connected to A, and at the moment we reach t_2, C was connected to B. So we can conclude that to reach t_2 via this path, two links must exist (not essentially at the same time). That is, a multi-hop communication link from A to C, denoted by $A \dashrightarrow C$, must exist to reach t_2. In general, to examine a property pre-conditioned by a multi-hop constraint over the topology, we look for a path in the CLTS along which the multi-hop relations are inferred.

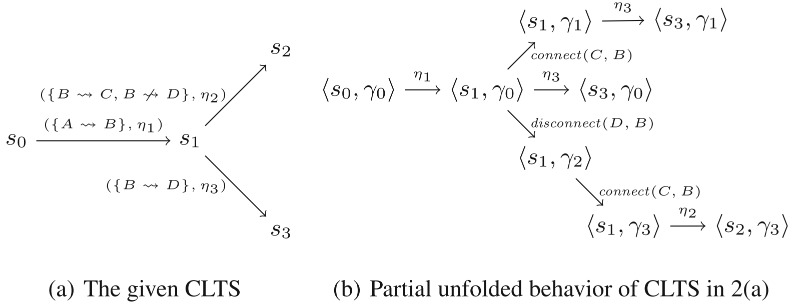
Let $T = \langle S, \Lambda, \rightarrow, s_0 \rangle$ be a CLTS. A *path* σ of T is a sequence of transitions $t_0(\mathcal{C}_0, \eta_0)t_1(\mathcal{C}_1, \eta_1)t_2 \ldots$ where $\forall i \geq 0\,((t_{i-1}(\mathcal{C}_i, \eta_i)t_i) \in \rightarrow)$. A path is said to be *maximal* if it either is infinite or ends in a deadlock state.

When a multi-hop relation is the pre-condition of a property, we are stating a set of single-hop links (leading to a multi-hop connection) required for a set of communications. Inversely, we can infer from the network constraints of such communications over a path that the multi-hop connection exists. To this aim, we determine multi-hop connections by collecting single-hop constraints along a path in a forward fashion using a set of computations over network constraints. The operator $\oplus : \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$ allows to gather information along a path in a CLTS. It merges connectivity information, where the second argument overwrites conflicting information of the first argument:

$$\mathcal{C}_1 \oplus \mathcal{C}_2 = \mathcal{C}_2 \cup \{p \mid \neg p \notin \mathcal{C}_2 \wedge p \in \mathcal{C}_1\}$$

where $\neg(\ell \rightsquigarrow \ell') = \ell \not\rightsquigarrow \ell'$ and $\neg(\ell \not\rightsquigarrow \ell') = \ell \rightsquigarrow \ell'$. This operator is left-associative and non-commutative; $\{\}$ is its identity element. Let $\oplus_{k=i}^{j} \mathcal{C}_k$ denote $(\ldots((\mathcal{C}_{k=i} \oplus \mathcal{C}_{k=i+1}) \oplus \mathcal{C}_{k=i+2}) \ldots \oplus \mathcal{C}_{k=j})$. We say $\xi \in \mathbb{C}$ *conforms* to $\mathcal{C}$ if ξ does not include (dis)connectivity information that contradicts $\mathcal{C}$, which can be formally tested by $\mathcal{C} \subseteq \mathcal{C} \oplus \xi$. Extending CLTSs with disconnectivity pairs allows to correctly update topology information gathered along a path when the communication behavior distinguishes lossy from disconnectivity by providing precise information in the labels (see Fig. 1(b) and 1(c)) in the case of unsuccessful communication. For instance, updating the connectivity information $\{A \rightsquigarrow B, C - B \rightsquigarrow C\}$ with $A \not\rightsquigarrow B$ results in $\{A \rightsquigarrow C, B \rightsquigarrow C, A \not\rightsquigarrow B\}$.

A path $t_0(\mathcal{C}_0, \eta_0)t_1(\mathcal{C}_1, \eta_1)t_2 \ldots$ is called $\mathcal{C}$-*path* if $\mathcal{C}_i$ conforms to $\mathcal{C}$ for all $i \geq 0$.



(a) The given CLTS (b) Partial unfolded behavior of CLTS in 2(a)

Fig. 2. Restricted mobility, achieved through restricted transition traversal

4.2 CACTL Syntax

To provide a logic to verify topology-dependent properties of MANET protocols, our modal path operator is parameterized with multi-hop constraints over the topology. This parameter specifies the pre-condition required for inspecting the property; if the pre-condition never holds, the property does not need to hold. Moreover, to verify properties of MANETs with regard to different mobility scenarios, the satisfaction relation is parameterized with single-hop constraints. This parameter expresses the (non-)existence of communication links and also restricts node mobility; nodes can only move in such a way that the specified links do not change. This is achieved by only traversing transitions that conform to the specified links. For instance, consider the CLTS in Fig. 2(a). We examine properties under the network constraint $\{B \not\rightsquigarrow C\}$, meaning that C is never in the transmission range of B. To this aim we should traverse transitions with network constraints $\mathcal{C}$ such that $\{B \not\rightsquigarrow C\} \subseteq \{B \not\rightsquigarrow C\} \oplus \mathcal{C}$ (like $s_0 \to s_1 \to s_3$ in the CLTS in Fig. 2(a)). This can be explained by partially unfolding the CLTS, as depicted in Fig. 2(b), with an initial topology γ_0 where $B \in \gamma_0(A)$, $D \in \gamma_0(B)$ and $C \notin \gamma_0(B)$. Three possible mobility scenarios of state $\langle s_1, \gamma_0 \rangle$ are shown: One moves C in the transmission range of B, while another moves D out and C in the transmission range of B. According to the mobility restriction, the resulting topologies γ_1 and γ_2 do not satisfy $\{B \rightsquigarrow C\}$. Therefore only the middle scenario to $\langle s_3, \gamma_0 \rangle$ is possible.

Our logic borrows its temporal operators from ACTLW: **AU**, **EU**, **AW**, and **EW**. Let $\eta \in Act_\tau$, $\ell, \ell' \in Loc$, $\mathcal{C} \in \mathbb{C}$. *Action formula* χ, *topology formula* μ, *state formula* ϕ (also called *CACTL formula*), and *path formula* ψ are defined by the grammars:

$$
\begin{array}{rl}
\chi ::= & true \mid \eta \mid \neg\chi \mid \chi \wedge \chi' \\
\mu ::= & true \mid \ell \dashrightarrow \ell' \mid \neg\mu \mid \mu \wedge \mu' \\
\phi ::= & true \mid \neg\phi \mid \phi \wedge \phi' \mid \mathbf{E}\psi \mid \mathbf{A}\psi \\
\psi ::= & \phi \;{}_{\{\chi\}}\mathbf{U}\,^{\mu}{}_{\{\chi'\}}\phi' \mid \phi \;{}_{\{\chi\}}\mathbf{W}\,^{\mu}{}_{\{\chi'\}}\phi'
\end{array}
$$

While action and state formulae are the same as in ACTLW, path formulae carry a condition over topologies. Intuitively, a path formula $\phi \;{}_{\{\chi\}}\mathbf{U}\,^{\mu}{}_{\{\chi'\}}\phi'$

specifies a path along which states satisfying property ϕ perform actions from χ, until the accumulated (dis)connectivity information along this path satisfies the topology formula μ, and a state satisfying property ϕ' is reached after an action from χ'. An infinite path that never stabilizes to a situation where μ is always satisfied, still satisfies this path formula if all its states satisfy ϕ and all its transitions are from χ. But if a path does stabilize to such a situation, then eventually a transition from χ' must lead to a state where ϕ' holds. Typically, μ could define that there is a path from node A to node B, and ϕ' could define that some information broadcast by A reaches B.

The path formula based on the unless (weak until) operator $\phi\ {}_{\{\chi\}}\mathbf{W}\ {}^{\mu}{}_{\{\chi'\}}\phi'$ specifies a path along which states satisfying property ϕ perform actions from χ at least as long as either μ is never satisfied or no state satisfying ϕ' is visited by an actions from χ'. We note that **EW** cannot readily be defined in terms of **AU** as opposed to CTL, due to actions of χ and χ' that should be visited to reach states satisfying ϕ and ϕ'.

4.3 CACTL Semantics

Let $\eta' \in Act_\tau$, $\zeta \in \mathbb{C}$, and $\langle S, \Lambda, \rightarrow, s_0\rangle$ a CLTS. Satisfaction under network constraint ζ of action formula χ by $\eta \in Act_\tau$ (written $\eta \models_\zeta \chi$), topology formula μ by network constraint $\mathcal{C}$ (written $\mathcal{C} \models_\zeta \mu$), state formula ϕ by state t (written $t \models_\zeta \phi$), or path formula ψ by maximal path σ (written $\sigma \models_\zeta \psi$), is inductively defined below. Let σ_i^s, $\sigma_i^{\mathcal{C}}$ and σ_i^η denote the i-th state, network constraint and action on path σ. With $\oplus_{k=1}^{\infty}\sigma_k^{\mathcal{C}} \not\models_\zeta \mu$ we mean that $\oplus_{k=1}^{m}\sigma_k^{\mathcal{C}} \models_\zeta \neg\mu$ for infinitely many $m \geq 1$.

$\eta \models_\zeta true$	always
$\eta \models_\zeta \eta'$	iff $\eta = \eta'$
$\eta \models_\zeta \neg\chi$	iff $\eta \not\models_\zeta \chi$
$\eta \models_\zeta \chi \wedge \chi'$	iff $\eta \models_\zeta \chi \wedge \eta \models_\zeta \chi'$
$\mathcal{C} \models_\zeta true$	always
$\mathcal{C} \models_\zeta \ell \dashrightarrow \ell'$	iff there are $\ell_0, \ldots, \ell_n \in Loc$ with $\ell_0 = \ell$, $\ell_n = \ell'$, and $\ell_i \rightsquigarrow \ell_{i+1} \in \zeta \oplus \mathcal{C}$ for all $i = 0, \ldots, n-1$
$\mathcal{C} \models_\zeta \neg\mu$	iff $\mathcal{C} \not\models_\zeta \mu$
$\mathcal{C} \models_\zeta \mu \wedge \mu'$	iff $\mathcal{C} \models_\zeta \mu \wedge \mathcal{C} \models_\zeta \mu'$
$t \models_\zeta true$	always
$t \models_\zeta \neg\phi$	iff $t \not\models_\zeta \phi$
$t \models_\zeta \phi \wedge \phi'$	iff $t \models_\zeta \phi \wedge t \models_\zeta \phi'$
$t \models_\zeta \mathbf{E}\psi$	iff there exists a maximal ζ-path σ such that $t = \sigma_0^s \wedge \sigma \models_\zeta \psi$
$t \models_\zeta \mathbf{A}\psi$	iff for all maximal ζ-paths σ, $t = \sigma_0^s \Rightarrow \sigma \models_\zeta \psi$
$\sigma \models_\zeta \phi\ {}_{\{\chi\}}\mathbf{U}\ {}^{\mu}{}_{\{\chi'\}}\phi'$	iff $\sigma_0^s \models_\zeta \phi$, and either $\oplus_{k=1}^{\infty}\sigma_k^{\mathcal{C}} \not\models_\zeta \mu$, $\forall j \geq 1\,(\sigma_j^s \models_\zeta \phi \wedge$ $(\sigma_j^\eta \models_\zeta \chi \wedge \zeta \subseteq \zeta \oplus \sigma_j^{\mathcal{C}}) \vee (\sigma_j^\eta = \tau \wedge \sigma_j^{\mathcal{C}} = \{\})))$ or there exists an $i \geq 1$

$$\text{such that } \sigma_i^s \models_{\zeta\oplus(\oplus_{k=1}^{i}\sigma_k^{\mathcal{C}})} \phi',\ \sigma_i^\eta \models_\zeta \chi',\ \zeta \subseteq \zeta \oplus \sigma_i^{\mathcal{C}},\ \oplus_{k=1}^{i}\sigma_k^{\mathcal{C}} \models_\zeta \mu, \text{ and}$$
$$\forall 1 \le j < i\,(\sigma_j^s \models_\zeta \phi \wedge ((\sigma_j^\eta \models_\zeta \chi \wedge \zeta \subseteq \zeta \oplus \sigma_j^{\mathcal{C}}) \vee (\sigma_j^\eta = \tau \wedge \sigma_j^{\mathcal{C}} = \{\})))$$
$$\sigma \models_\zeta \phi\ {}_{\{\chi\}}\mathbf{W}\ {}^{\mu}{}_{\{\chi'\}}\phi' \quad \text{iff } \sigma \models_\zeta \phi\ {}_{\{\chi\}}\mathbf{U}\ {}^{\mu}{}_{\{\chi'\}}\phi', \text{ or } \sigma_0^s \models_\zeta \phi \text{ and } \forall j \ge 1\,(\sigma_j^s \models_\zeta \phi$$
$$\wedge ((\sigma_j^\eta \models_\zeta \chi \wedge \zeta \subseteq \zeta \oplus \sigma_j^{\mathcal{C}}) \vee (\sigma_j^\eta = \tau \wedge \sigma_j^{\mathcal{C}} = \{\})))$$

We use $\models$ to denote $\models_{\{\}}$ and ${}_{\{\chi\}}\mathbf{U}\ {}_{\{\chi'\}}$ to denote ${}_{\{\chi\}}\mathbf{U}\ {}^{true}{}_{\{\chi'\}}$. We remark that the semantics of the until operator is somewhat different from CTL: $\mathbf{E}(\phi_{\{\chi\}}\mathbf{U}_{\{\chi'\}}\phi')$ in ACTLW (similar to ACTL) requires to perform at least one action from $\{\chi'\}$ to reach a state satisfying ϕ', while $\mathbf{E}(\phi\mathbf{U}\phi')$ in CTL is satisfied if the first state satisfies ϕ'. Furthermore, our semantics explicitly distinguishes silent actions τ (over all possible topologies) and visible actions (similar to ACTL, in contrast to ACTLW).

For example, the state formula $\mathbf{A}(true\ {}_{\{\tau\vee init\}}\mathbf{U}\ {}^{A\dashrightarrow B}{}_{\{get\}}true)$ indicates that if there exists a path from A to B, the action *init* is followed by action *get*, after some communication (specified by τ). E.g., the path

$$\mathcal{M}_0 \xrightarrow{(\{\},init)} \mathcal{M}_1 \xrightarrow{(\{A\rightsquigarrow C\ -\ A\not\rightsquigarrow B,D\},\tau)} \mathcal{M}_2 \xrightarrow{(\{C\rightsquigarrow D\ -\ C\not\rightsquigarrow A,B\},\tau)} \mathcal{M}_3$$
$$\xrightarrow{(\{D\rightsquigarrow B\ -\ D\not\rightsquigarrow C\},\tau)} \mathcal{M}_4 \xrightarrow{(\{\},get)} \mathcal{M}_5 \rightarrow \ldots$$

satisfies $(true\ {}_{\{\tau\vee init\}}\mathbf{U}\ {}^{A\dashrightarrow B}{}_{\{get\}}true)$ with no restriction on mobility of nodes ($\zeta = \{\}$), since *get* is observed after passing transitions with labels of *init* and τ, and the accumulated network constraints over these transitions, $\{\} \oplus \{A \rightsquigarrow C\ -\ A \not\rightsquigarrow B, D\} \oplus C \rightsquigarrow D\ -\ C \not\rightsquigarrow A, B\} \oplus \{D \rightsquigarrow B\ -\ D \not\rightsquigarrow C\} \oplus \{\}$ induces that $A \dashrightarrow B$ via C and D.

4.4 CACTL Model Checking

We adapt the CTL model checking algorithm (see [6]) to CACTL. Model checking a CACTL formula φ under $\zeta \in \mathbb{C}$ starts with smallest sub-formulae and works outwards toward φ. The model checking will operate by adding to each state a set of labels of the form $\langle\phi, \Omega, O\rangle$, where ϕ is a subformula of φ, Ω a set of (dis)connectivity pairs, not necessarily well-formed, and O a set of *topology obligations*. The set Ω maintains the history of links experienced during exploration of φ, and is helpful in the verification of state formulae based on (weak) until operators. A topology obligation is a pair of a topology formula μ and a network constraint $\mathcal{C}$. A topology obligation $\langle\mu, \mathcal{C}\rangle$ is said to be satisfied when $\mathcal{C} \models_\zeta \mu$. Initially all states are labeled by $\langle true, \emptyset, \emptyset\rangle$. A state satisfies φ iff at the end it includes a label $\langle\varphi, \Omega, O\rangle$ with the topology obligations in O all satisfied. The pseudo code of the model checking algorithm's backbone is given in Fig. 3.

We explain the idea of procedure *CheckEU* for the **EU** operator; other CACTL operators can be dealt with in a similar way. The pseudo code of this procedure is given in Appendix A. For simplicity we assume that CLTSs are deadlock-free. We extend the application of $\oplus$ to topology obligations: $\mathcal{C} \oplus O = \{\langle\mu, \mathcal{C} \oplus \mathcal{C}'\rangle \mid \langle\mu, \mathcal{C}'\rangle \in O\}$. Two possible cases should be examined. In the first case, state s satisfies formula $\mathbf{E}(\phi\ {}_{\{\chi\}}\mathbf{U}\ {}^{\mu}{}_{\{\chi'\}}\phi')$ if there exists a path from s consisting of states satisfying property ϕ under ζ and actions from χ, until a state satisfying ϕ' under $\zeta \oplus \xi$ is reached after an action from χ' and ξ induces μ, where ξ is the accumulated (dis)connectivity information along this path. To check this case, we move backward starting from the states where ϕ' holds under ζ, first over a transition with an action from χ', and then

Procedure $ModelCheck(\varphi, \zeta)$
Initially set the labels of all states to $\{\langle true, \emptyset, \emptyset\rangle\}$;
repeat
  let ϕ be the next innermost formula of φ;
  switch ϕ **do**
    case $\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi')$
      $ModelCheck(\phi, \zeta)$;
      $ModelCheck(\phi', \zeta)$;
      $CheckEU(\phi, \chi, \phi', \chi', \mu, \zeta)$;
    case $\phi \wedge \phi'$
      $ModelCheck(\phi, \zeta)$;
      $ModelCheck(\phi', \zeta)$;
      $CheckAnd(\phi, \phi', \zeta)$;
    ...
  endsw
until $\phi = \varphi$;
if $\exists\langle\varphi, \Omega, O\rangle \in label(s_0) \wedge \forall\langle\mu, \mathcal{C}\rangle \in O \cdot \mathcal{C} \models_\zeta \mu$ **then return** $true$;
else return $false$;

Fig. 3. Model checking algorithm

over transitions with an action from χ, passing over states where ϕ holds under ζ. We record the status of links encountered during backward exploration of executions (note that these links conform to ζ). To ensure conformability of the links recorded for ϕ' to $\zeta \oplus \xi$, we incrementally check conformability of these links to the partial of ξ being formed in the backward exploration. Since the yet unknown ξ should induce μ, we initially include topology obligation $\langle\mu, \{\}\rangle$ in the state label; its network constraint is incrementally updated while moving backward. Furthermore, we record the topology obligation generated during exploration of ϕ and ϕ'. To ensure ϕ' holds under $\zeta \oplus \xi$, we incrementally update its recorded topology obligation while moving backward.

Let Ω and Ω' contain the links that occurred over executions during exploration of ϕ and ϕ', and O and O' the topology obligations generated during exploration of ϕ and ϕ' (under ζ), respectively. Since first ϕ and only after that ϕ' needs to hold, these sets can be kept separate. The sets Ω and Ω' may contain conflicting conditions, and even if Ω' conforms to the partial of ξ, $\Omega \cup \Omega'$ may not. To check conformability of Ω' to and update O' with partial ξ, we postpone mixing these sets until the end, and exploit a senary labeling $\langle\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi'), \Omega, \Omega', \mathcal{C}, O, O'\rangle$. Let $\mathcal{C}$ be the accumulated value of network constraints over the traversed execution path. By moving backward over a $(\mathcal{C}, \eta)$-transition (where $\mathcal{C}$ conforms to ζ and η satisfies χ') from the state labeled with $\langle\phi', \Omega', O'\rangle$ to the state labeled with $\langle\phi, \Omega, O\rangle$, we add the label $\langle\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi'), \Omega \cup \mathcal{C}, \Omega', \mathcal{C}, O, \{\langle\mu, \mathcal{C}\rangle\} \cup \mathcal{C} \oplus O'\rangle$ to states labeled with $\langle\phi, \omega, O\rangle$, if Ω' conforms to $\mathcal{C}$; it should be noted that $\mathcal{C}$ is added to Ω (and Ω conforms to ζ), O' is updated with $\mathcal{C}$, and the obligation $\{\langle\mu, \mathcal{C}\rangle\}$ ($\mathcal{C} \oplus \{\langle\mu, \{\}\rangle\}$) is generated during model checking of $\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi')$. At the end of model checking, the senary labels $\langle\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi'), \Omega, \Omega', \mathcal{C}, O, O'\rangle$ are replaced by $\langle\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi'), \Omega \cup \Omega', O \cup O'\rangle$.

Next we continue moving backward from the states with labels of the form $\langle\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi'), \Omega, \Omega', \mathcal{C}', O, O'\rangle$ over $(\{\}, \tau)$ or $(\mathcal{C}, \eta)$-transitions of which their action satisfies χ and their network constraint conforms to ζ, to reach states labeled by $\langle\phi, \Omega'', O''\rangle$ for some Ω'' and O''. We add the label $\langle\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi'), \Omega \cup \Omega'' \cup$

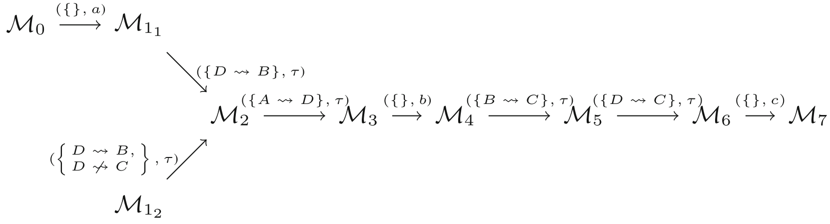


Fig. 4. The CLTS to be checked for $\mathbf{E}(true\ _{\{a\vee\tau\}}\mathbf{U}\ ^{A\dashrightarrow B}{}_{\{b\}}\mathbf{E}(true\ _{\{a\vee\tau\}}\mathbf{U}\ ^{A\dashrightarrow C}{}_{\{c\}}\ true))$

Table 1. Labels of states in Fig. 4 while checking formula $\varphi_2 \equiv \mathbf{E}(\phi\ _{\{a\vee\tau\}}\mathbf{U}\ ^{A\dashrightarrow B}{}_{\{b\}}\varphi_1)$, where $\varphi_1 \equiv \mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{A\dashrightarrow C}{}_{\{c\}}\phi)$ and $\phi \equiv true$

Steps	Actions
1	$\langle true, \emptyset, \emptyset\rangle$ are added to $\mathcal{M}_{0,1_1,1_2,2-7}$
2	$L_1 \equiv \langle \varphi_1, \emptyset, \emptyset, \{\}, \emptyset, \{\langle A \rightsquigarrow C, \{\}\rangle\}\rangle$ is added to $\mathcal{M}_6$
3	$L_2 \equiv \langle \varphi_1, \{D \rightsquigarrow C\}, \emptyset, \{D \rightsquigarrow C\}, \emptyset, \{\langle A \dashrightarrow C, \{D \rightsquigarrow C\}\rangle\}\rangle$ is added to $\mathcal{M}_5$
4	$L_3 \equiv \langle \varphi_1, \{B \rightsquigarrow C, D \rightsquigarrow C\}, \emptyset, \{D \rightsquigarrow C, B \rightsquigarrow C\}, \emptyset, \{\langle A \dashrightarrow C, \{D \rightsquigarrow C, B \rightsquigarrow C\}\rangle\}\rangle$ is added to $\mathcal{M}_4$
5	L_1 is replaced by $\langle \varphi_1, \emptyset, \{\langle A \dashrightarrow C, \{\}\rangle\}\rangle$ in $\mathcal{M}_6$
6	L_2 is replaced by $\langle \varphi_1, \{D \rightsquigarrow C\}, \{\langle A \dashrightarrow C, \{D \rightsquigarrow C\}\rangle\}\rangle$ in $\mathcal{M}_5$
7	L_3 is replaced by $\langle \varphi_1, \{B \rightsquigarrow C, D \rightsquigarrow C\}, \{\langle A \dashrightarrow C, \{B \rightsquigarrow C, D \rightsquigarrow C\}\rangle\}\rangle$ in $\mathcal{M}_4$
8	$L_4 \equiv \langle \varphi_2, \emptyset, \{B \rightsquigarrow C, D \rightsquigarrow C\}, \{\}, \emptyset, \{\langle A \dashrightarrow C, \{B \rightsquigarrow C, D \rightsquigarrow C\}\rangle, \langle A \dashrightarrow B, \{\}\rangle\}\rangle$ is added to $\mathcal{M}_3$
9	$L_5 \equiv \langle \varphi_2, \{A \rightsquigarrow D\}, \{B \rightsquigarrow C, D \rightsquigarrow C\}, \{A \rightsquigarrow D\}, \emptyset, \{\langle A \dashrightarrow C, \{A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\}\rangle,$ $\langle A \dashrightarrow B, \{A \rightsquigarrow D\}\rangle\}\rangle$ is added to $\mathcal{M}_2$
10	$L_6 \equiv \langle \varphi_2, \{D \rightsquigarrow B, A \rightsquigarrow D\}, \{B \rightsquigarrow C, D \rightsquigarrow C\}, \{D \rightsquigarrow B, A \rightsquigarrow D\}, \emptyset,$ $\{\langle A \dashrightarrow C, \{D \rightsquigarrow B, A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\}\rangle, \langle A \dashrightarrow B, \{D \rightsquigarrow B, A \rightsquigarrow D\}\rangle\}\rangle$ is added to $\mathcal{M}_{1_1}$ and $\mathcal{M}_0$
11	L_4 is replaced by $\langle \varphi_2, \{B \rightsquigarrow C, D \rightsquigarrow C\}, \{\langle A \dashrightarrow C, \{B \rightsquigarrow C, D \rightsquigarrow C\}\rangle, \langle A \dashrightarrow B, \{\}\rangle\}\rangle$ in $\mathcal{M}_3$
12	L_5 is replaced by $\langle \varphi_2, \{A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\}, \{\langle A \dashrightarrow C, \{A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\}\rangle,$ $\langle A \dashrightarrow B, \{A \rightsquigarrow D\}\rangle\}\rangle$ in $\mathcal{M}_2$
13	L_6 is replaced by $\langle \varphi_2, \{D \rightsquigarrow B, A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\},$ $\{\langle A \dashrightarrow C, \{D \rightsquigarrow B, A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\}\rangle, \langle A \dashrightarrow B, \{D \rightsquigarrow B, A \rightsquigarrow D\}\rangle\}\rangle$ in $\mathcal{M}_1$ and $\mathcal{M}_0$

$\mathcal{C}, \Omega', \mathcal{C} \oplus \mathcal{C}', O \cup O'', \mathcal{C} \oplus O'\rangle$ to these states, if Ω' conforms to $\mathcal{C} \oplus \mathcal{C}'$. We continue moving backward until no new label is added to the states.

As an example, we verify $\mathbf{E}(true\ _{\{a\vee\tau\}}\mathbf{U}\ ^{A\dashrightarrow B}{}_{\{b\}}\mathbf{E}(true\ _{\{a\vee\tau\}}\mathbf{U}\ ^{A\dashrightarrow C}{}_{\{c\}}\ true))$ under $\{\}$ over the CLTS given in Fig. 4. States are initially labeled by $\langle true, \emptyset, \emptyset\rangle$. Table 1 includes the labels given to the states in each step; first we label states $\mathcal{M}_7$ to $\mathcal{M}_4$ for the inner until operator, and then we label states $\mathcal{M}_4$ to $\mathcal{M}_0$ for the outer until operator. State $\mathcal{M}_{1_2}$ is only labeled by $\langle true, \emptyset, \emptyset\rangle$ and cannot be labeled further, because the set of links encountered during exploration of the inner until formula, i.e. $\{B \rightsquigarrow C, D \rightsquigarrow C\}$, does not conform to $\{D \rightsquigarrow B, D \not\rightsquigarrow C\}$. State $\mathcal{M}_0$ includes the label $\langle \varphi_2, \{D \rightsquigarrow B, A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\}, \{\langle A \dashrightarrow C, \{D \rightsquigarrow B, A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\}\rangle, \langle A \dashrightarrow B, \{D \rightsquigarrow B, A \rightsquigarrow D\}\rangle\}\rangle$, so it satisfies φ_2 (under $\{\}$), since $\{D \rightsquigarrow B, A \rightsquigarrow D, B \rightsquigarrow C, D \rightsquigarrow C\} \models A \dashrightarrow C$ and $\{D \rightsquigarrow B, A \rightsquigarrow D\} \models A \dashrightarrow B$. The topology formula $A \dashrightarrow C$ in the inner until formula is satisfied after the network

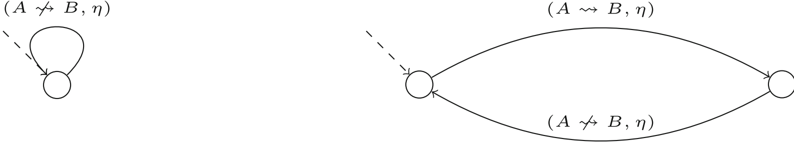


Fig. 5. Two examples of infinite execution paths for which the accumulated network constraints never induce $A \dashrightarrow B$ permanently

constraints $A \rightsquigarrow D$ and $D \rightsquigarrow B$ update their corresponding topology obligation while moving backward to model check the outer formula.

In the second case, s satisfies $\mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi')$ if there exists a path from s along which the states satisfy ϕ, the actions are from χ, and the accumulated (dis)connectivity information never induces μ permanently (see Fig. 5 for two simple examples). To check the occurrence of this case, we decompose the CLTS into non-trivial strongly connected components (SCCs), meaning that they contain at least one edge.

We first restrict to states that include a label $\langle \phi, \Omega, O\rangle$ for some Ω and O, and to $(\{\}, \tau)$ and $(\mathcal{C}, \eta)$-transitions where η satisfies χ and $\mathcal{C}$ conforms to ζ. Next, we partition the new CLTS into SCCs using the algorithm explained in [2]. We initially move backward in an SCC over $(\mathcal{C}, \eta)$-transitions, and for any $\langle \phi, \Omega_1, O_1\rangle \in \mathit{label}(s)$ and $\langle \phi, \Omega_2, O_2\rangle \in \mathit{label}(t)$, where s and t are origin and destination of transition, we add the label $\langle \mathbf{E}(\phi\ _{\{\chi\}}\mathbf{U}\ ^{\mu}{}_{\{\chi'\}}\phi'), \Omega_1 \cup \Omega_2 \cup \mathcal{C}, \emptyset, \mathcal{C}, O_1 \cup O_2, \{\langle \neg\mu, \mathcal{C}\rangle\}\rangle$ to s. The obligation $\langle \neg\mu, \mathcal{C}\rangle$ indicates that the accumulated value of network constraints does not induce μ. Then, similar to first case, we move backward out of the SCC until no new label is added to the states, and at the end we replace senary labels by triples.

5 Protocol Analysis with CACTL

To illustrate the expressiveness of CACTL in the analysis of MANETs, we specify properties for two important classes of protocols, namely routing and leader election.

The most fundamental error in routing protocol operations is failure to route correctly. The correct operation of MANET routing protocols is defined as follows [26]: *If from some point in time on there exists a path between two nodes, then the protocol must be able to find some path between the nodes. Furthermore, when a path has been found, and for the time it stays valid, it must be possible to send packets along the path from the source node to the destination node.* To verify the first part of property, let $\mathit{init}(\mathit{src})$ and $\mathit{found}(\mathit{src})$ indicate initialization and completion respectively of the route discovery protocol in a node with address src for a specific address dst. The property "whenever there exists a path from src to dst and from dst to src, each $\mathit{init}(\mathit{src})$ is proceeded by its corresponding $\mathit{found}(\mathit{src})$", for $\mathit{scr} \in \{A, C\}$ and $\mathit{dst} = B$, is specified by the CACTL formula

$$\mathbf{A}(\mathit{true}\ _{\{\tau \vee \mathit{init}(A) \vee \mathit{init}(C)\}}\mathbf{U}\ ^{A \dashrightarrow B \wedge B \dashrightarrow A}{}_{\{\mathit{found}(A)\}}\mathit{true}) \wedge$$
$$\mathbf{A}(\mathit{true}\ _{\{\tau \vee \mathit{init}(C) \vee \mathit{init}(C)\}}\mathbf{U}\ ^{C \dashrightarrow B \wedge B \dashrightarrow C}{}_{\{\mathit{found}(C)\}}\mathit{true})$$

where τ abstracts away from communications between nodes. By model checking the CLTS model of a MANET in which the nodes deploy a routing protocol, we can verify

this property with respect to arbitrary topology changes. This property was examined in [7] for the AODV protocol using CTL.

To verify the second part of the property, let *insert*(*src*) and *delivery*(*src*) indicate submission and arrival of a data over the route found beforehand at *src* and *dst* respectively. The property "whenever there exists paths from *src* to *dst* and from *dst* to *src*, when a route is found from *src* to *dst*; and while this path is valid each *insert* is followed by *deliver*", for $src = C$ and $dst = B$, is specified by the CACTL formula

$$\mathbf{A}(true\ _{\{init(C)\vee init(A)\vee\tau\}}\mathbf{U}\ ^{C\dashrightarrow B\wedge B\dashrightarrow C}{}_{\{found(C)\}}(\mathbf{A}(true\ _{\{insert\vee\tau\}}\mathbf{U}\ ^{true}{}_{\{deliver\}}true)))$$

The outer until formula looks for all maximal paths in which *found* is performed after *init* while the accumulated network constraints ξ induce $C \dashrightarrow B$ and $B \dashrightarrow C$ (or this topology formula is never induced), and by *found* reach a state of which all maximal ξ-paths satisfy the inner until path formula. The network constraints over a ξ-path do not violate the single-hop (dis)connectivity pairs in ξ. It can be said that (dis)connectivity pairs are frozen, and consequently the route from *src* to *dst* is still valid.

Classical leader election algorithms aim at electing a unique leader from a fixed set of nodes. In the context of MANETs such protocols should consider arbitrary topology changes, and aim at finding a unique leader which is the most-valued node within a connected component [25]. Let *leader*(*id*, *lid*) indicate that a node with address *id* has found its leader with address *lid*, and let node A be the most-valued node. We can investigate correctness of such a leader election algorithm with the CACTL formula

$$\mathbf{A}(true\ _{Act_\Lambda}\mathbf{U}\ ^{\substack{A\dashrightarrow B\wedge A\dashrightarrow C\wedge\\ B\dashrightarrow A\wedge B\dashrightarrow C\wedge\\ C\dashrightarrow A\wedge C\dashrightarrow B}}{}_{\{leader(A,A),leader(B,A),leader(C,A)\}}true)$$

It expresses that in any connected component containing nodes A, B and C, eventually A is chosen as the leader. Being in the same connected component is indicated by the existence of multi-hop paths among them. We can also investigate scenarios in which two disconnected components merge together with the help of a CACTL formula like

$$\begin{aligned}\mathbf{A}(true\ _{Act_\Lambda}\mathbf{U}\ ^{\substack{A\dashrightarrow B\wedge B\dashrightarrow A\wedge\\ C\dashrightarrow D\wedge D\dashrightarrow C\wedge\\ \neg A\dashrightarrow C\wedge\neg A\dashrightarrow D\wedge\\ \neg B\dashrightarrow C\wedge\neg B\dashrightarrow D}}{}_{\{leader(A,A),leader(B,A),leader(D,C)\}}(\\ \mathbf{A}(true\ _{Act_\Lambda}\mathbf{U}\ ^{C\dashrightarrow B\wedge B\dashrightarrow C}{}_{\{leader(A,A),leader(B,A),leader(D,A),leader(C,A)\}}true))\end{aligned}$$

meaning that nodes A, B and nodes C, D belong to the same component with leader A and C respectively; if these two components get connected via B and C, then they will eventually converge to the same leader, i.e. A.

6 Branching Network Bisimilarity

We define a novel notion of branching network bisimilarity that induces the same identification of CLTSs as our logical framework.

Definition 1. *Let* $\langle S, \Lambda, \rightarrow, s_0\rangle$ *be a CLTS. States* $r, s \in S$ *are logically equivalent, denoted by* $r \sim_L s$, *iff* $\forall\zeta \in \mathbb{C}\ \forall\varphi \in CACTL\ (r \models_\zeta \varphi \Leftrightarrow s \models_\zeta \varphi)$.

Intuitively, equivalent states in a CLTS exhibit the same behavior for any topology. This behavior includes communication and internal actions. Communication actions

carry a message and the address of the sender, which can be abstracted into the unknown address ?. Two equivalent states must match on every internal action, receive action, and send action with a known address. A send action with unknown address can be mimicked by a send action with either a known or unknown address. Let $\Longrightarrow$ denote the reflexive-transitive closure of τ-transitions, over all possible topologies.

Definition 2. *A binary relation $\mathcal{R}$ on states in a CLTS is a* branching network simulation *if $t_1 \mathcal{R} t_2$ and $t_1 \xrightarrow{(\mathcal{C},\eta)} t_1'$ implies that:*

- *either $(\mathcal{C},\eta)$ is $(\{\},\tau)$, and $t_1' \mathcal{R} t_2$; or*
- *there are t_2' and t_2'' such that $t_2 \Longrightarrow t_2' \xrightarrow{(\mathcal{C},\eta)} t_2''$, where $t_1 \mathcal{R} t_2'$ and $t_1' \mathcal{R} t_2''$; or*
- *$\eta \equiv nsnd(\mathfrak{m}, ?)$, and there are t_2', t_2'' and ℓ such that $t_2 \Longrightarrow t_2' \xrightarrow{(\mathcal{C}[\ell/?],nsnd(\mathfrak{m},\ell))} t_2''$, where $t_1 \mathcal{R} t_2'$ and $t_1' \mathcal{R} t_2''$.*

$\mathcal{R}$ is a branching network bisimulation *if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are branching network simulations. Two terms t_1 and t_2 are branching network bisimilar, denoted by $t_1 \simeq_b t_2$, if $t_1 \mathcal{R} t_2$ for some branching network bisimulation relation $\mathcal{R}$.*

Theorem 1. *$\simeq_b$ is an equivalence relation.*

This theorem can be proved in a similar fashion as for branching computed network bisimilarity in [9]. As said, branching network bisimilarity and the equivalence relation induced by CACTL coincide. This can be proved for CLTSs with so-called bounded-nondeterminism following the approach of [4]. The result can be lifted to general CLTSs in the same vein as [19], by resorting to infinitary logics (see [11] for the proof).

Theorem 2. *Let $\langle S, \Lambda, \rightarrow, s_0 \rangle$ be a CLTS. For any $r, s \in S$, $r \simeq_b s$ iff $r \sim_L s$.*

7 Conclusion and Future Work

We introduced the branching-time temporal logic CACTL, interpreted over CLTSs, to reason about topology-dependent behavior of MANET protocols. We can investigate scenarios like *after a route found* and *after two disconnected components merged* with the help of multi-hop constraints over topologies, which are specified as a part of path operators in our logic. Advantages of our approach are flexibility in verifying topology-dependent behavior (without changing the model), and restricting the generality of mobility. By nesting until operators, a specific path can be found with the help of topology constraints (without a need to specify how a topology constraint should be inferred), and then fixed for further exploration. The (dis)connectivity information in CLTS transitions makes it possible to restrict the generality of mobility as desired. By contrast, in approaches like [7], the inferences leading to the establishment of topology constraints should be embedded in the specification. Existing approaches to model mobility either are insusceptible to model checking [7,13,23], or require separate modeling of mobility [8]. The logic in [21] does not support verification of topology-dependent behavior.

A model checker for CACTL is being implemented, using the rewrite logic Maude. We also intend to verify real-world MANET protocols.

References

1. Bhargavan, K., Obradovid, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. Journal of the ACM 49(4), 538–576 (2002)
2. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
3. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
4. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. Journal of the ACM 42(2), 458–487 (1995)
5. De Renesse, R., Aghvami, A.H.: Formal verification of ad-hoc routing protocols using SPIN model checker. In: MELECON, pp. 1177–1182. IEEE (2004)
6. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2001)
7. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 295–315. Springer, Heidelberg (2012)
8. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Automated analysis of AODV using Uppaal. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 173–187. Springer, Heidelberg (2012)
9. Ghassemi, F., Fokkink, W., Movaghar, A.: Equational reasoning on mobile ad hoc networks. Fundamenta Informaticae 103, 1–41 (2010)
10. Ghassemi, F., Fokkink, W., Movaghar, A.: Verification of mobile ad hoc networks: An algebraic approach. Theoretical Computer Science 412(28), 3262–3282 (2011)
11. Ghassemi, F., Ahmadi, S., Fokkink, W., Movaghar, A.: Model Checking MANETs with Arbitrary Mobility. In: Arbab, F., Sirjani, M. (eds.) FSEN 2013. LNCS, vol. 8161, pp. 214–228. Springer, Heidelberg (2013)
12. Godskesen, J.C.: Observables for mobile and wireless broadcasting systems. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 1–15. Springer, Heidelberg (2010)
13. Godskesen, J.C.: A calculus for mobile ad hoc networks. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 132–150. Springer, Heidelberg (2007)
14. Kouzapas, D., Philippou, A.: A process calculus for dynamic networks. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 213–227. Springer, Heidelberg (2011)
15. McIver, A., Fehnker, A.: Formal Techniques for Analysis of Wireless Network. In: ISoLA. LNCS vol. 6722, pp. 263–270. IEEE (2006)
16. Meolic, R., Kapus, T., Brezocnik, Z.: ACTLW - An action-based computation tree logic with unless operator. Information Sciences 178(6), 1542–1557 (2008)
17. Merro, M.: An observational theory for mobile ad hoc networks. In: MFPS XXIII. ENTCS, vol. 173, pp. 275–293. Elsevier (2007)
18. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. In: MFPS XXII. ENTCS, vol. 158, pp. 331–353. Elsevier (2006)
19. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
20. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theoretical Computer Science 367(1), 203–227 (2006)
21. Nanz, S., Nielson, F., Nielson, H.: Static analysis of topology-dependent broadcast networks. Information and Computation 208(2), 117–139 (2010)

22. Perkins, C.E., Belding-Royer, E.M.: Ad-hoc on-demand distance vector routing. In: WMCSA, pp. 90–100. IEEE (1999)
23. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 296–314. Springer, Heidelberg (2008)
24. van Glabbeek, R., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3), 555–600 (1996)
25. Vasudevan, S., Kurose, J., Towsley, D.: Design and analysis of a leader election algorithm for mobile ad hoc networks. In: ICNP, pp. 350–360. IEEE Computer Society (2004)
26. Wibling, O., Parrow, J., Pears, A.: Automatized verification of ad hoc routing protocols. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 343–358. Springer, Heidelberg (2004)
27. Wibling, O., Parrow, J., Pears, A.: Ad hoc routing protocol verification through broadcast abstraction. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 128–142. Springer, Heidelberg (2005)

A Pseudo Code of Procedure *CheckEU*

Procedure *CheckEU* $(\phi_1, \chi_1, \phi_2, \chi_2, \mu, \zeta)$
$T' := \{s \mid \langle \phi_2, -, - \rangle \in \mathit{label}(s)\}$; `//--the first case--`
$T := \emptyset$ and let $\varphi \equiv \mathbf{E}(\phi_1 \,_{\{\chi_1\}}\mathbf{U}\,^{\mu}_{\{\chi_2\}}\phi_2)$;
forall the t *such that* $(t, (\mathcal{C}, \eta), s) \in \rightarrow$ *and* $\eta \models \chi_2$ *and* $\zeta \subseteq \zeta \oplus \mathcal{C}$ **do**
  forall the $\langle \phi_1, \Omega_1, O_1 \rangle \in \mathit{label}(t)$ **do**
    forall the $\langle \phi_2, \Omega_2, O_2 \rangle \in \mathit{label}(s)$ *such that* $\mathcal{C} \subseteq \mathcal{C} \oplus \Omega_2$ **do**
      $\mathit{label}(t) := \mathit{label}(t) \cup \{\langle \varphi, \mathcal{C} \cup \Omega_1, \Omega_2, \mathcal{C}, O_1, \{\langle \mu, \mathcal{C} \rangle\} \cup \mathcal{C} \oplus O_2 \rangle\}$;
      $T := T \cup \{t\}$;
while $T \neq \emptyset$ **do**
  choose $s \in T$ and $T := T \setminus \{s\}$;
  forall the $(t, (\mathcal{C}, \eta), s) \in \rightarrow$ *and* $((\eta \models \chi_1 \wedge \zeta \subseteq \zeta \oplus \mathcal{C}) \vee (\mathcal{C} = \{\} \wedge \eta = ?))$ **do**
    forall the $\langle \phi_1, \Omega, O \rangle \in \mathit{label}(t)$ **do**
      forall the $\langle \varphi, \Omega_1, \Omega_2, \mathcal{C}', O_1, O_2 \rangle \in \mathit{label}(s)$ *s.t.* $\mathcal{C} \oplus \mathcal{C}' \subseteq \mathcal{C} \oplus \mathcal{C}' \oplus \Omega_2$ **do**
        $\mathit{newLabel} = \mathit{label}(t) \cup \{\langle \varphi, \mathcal{C} \cup \Omega \cup \Omega_1, \Omega_2, \mathcal{C} \oplus \mathcal{C}', O \cup O_1, \mathcal{C} \oplus O_2 \rangle\}$;
        if $\mathit{newLabel} \setminus \mathit{label}(t) \neq \emptyset$ **then**
          $\mathit{label}(t) := \mathit{newLabel}$;
          $T := T \cup \{t\}$;
        endif
endwhile
$T := \{s \mid \langle \varphi, -, -, -, -, - \rangle \in \mathit{label}(s)\}$;
forall the $s \in T$ **do**
  forall the $\langle \varphi, \Omega_1, \Omega_2, \mathcal{C}, O_1, O_2 \rangle \in \mathit{label}(s)$ **do**
    $\mathit{label}(s) := \mathit{label}(s) \setminus \{\langle \varphi, \Omega_1, \Omega_2, \mathcal{C}, O_1, O_2 \rangle\}$;
    $\mathit{label}(s) := \mathit{label}(s) \cup \{\langle \varphi, \Omega_1 \cup \Omega_2, O_1 \cup O_2 \rangle\}$;

$S' = \{s \mid \langle \phi_1, -, - \rangle \in label(s)\}$; `//--the second case--`
$\rightarrow' = \{(t, (\mathcal{C}, \eta), s) \mid (t, (\mathcal{C}, \eta), s) \in \rightarrow \wedge ((\eta \models \chi_1 \wedge \zeta \subseteq \zeta \oplus \mathcal{C}) \vee (\mathcal{C} = \{\} \wedge \eta = ?)) \wedge s, t \in S'\}$;
$SCC := \{SC \mid SC$ is a non-trivial SCC of CLTS$\langle S', \Lambda, \rightarrow', s_0 \rangle\}$;
$T' := \bigcup_{SC \in SCC} \{s \mid s \in SC\}$;
forall the $s \in T'$ *do* **do** `/* initializing states on SCCs */`
  forall the $(t, (\mathcal{C}, \eta), s) \in \rightarrow'$ *and* $\eta \models \chi_1 \wedge \zeta \subseteq \zeta \oplus \mathcal{C}$ **do**
    forall the $\langle \phi_1, \Omega_1, O_1 \rangle \in label(t)$ **do**
      forall the $\langle \phi_1, \Omega_2, O_2 \rangle \in label(s)$ **do**
        $label(t) := label(t) \cup \{\langle \varphi, \mathcal{C} \cup \Omega_1 \cup \Omega_2, \emptyset, \mathcal{C}, O_1 \cup O_2, \{\langle \neg\mu, \mathcal{C} \rangle\} \rangle\}$;
T := T';
while $T \neq \emptyset$ **do** `/* finding the accumulated network constraints in each SCC and next moving out of SCCs */`
  choose $s \in T$;
  $T := T \setminus \{s\}$;
  forall the $t \in S'$ *such that* $(t, (\mathcal{C}, \eta), s) \in \rightarrow'$ **do**
    forall the $\langle \phi_1, \Omega, O \rangle \in label(t)$ **do**
      forall the $\langle \varphi, \Omega_1, \Omega_2, \mathcal{C}', O_1, O_2 \rangle \in label(s)$ **do**
        $newLabel = label(t) \cup \{\langle \varphi, \mathcal{C} \cup \Omega \cup \Omega_1, \Omega_2, \mathcal{C}\mathcal{C}', O \cup O_1, \mathcal{C} \oplus O_2 \rangle\}$;
        if $newLabel \setminus label(t)$ **then**
          $label(t) := newLabel$;
          $T := T \cup \{t\}$;
        endif
endwhile
$T := \{s \mid \langle \varphi, -, -, -, - \rangle \in label(s)\}$;
forall the $s \in T$ **do**
  forall the $\langle \varphi, \Omega_1, \Omega_2, \mathcal{C}, O_1, O_2 \rangle \in label(s)$ **do**
    $label(s) := label(s) \setminus \{\langle \varphi, \Omega_1, \Omega_2, \mathcal{C}, O_1, O_2 \rangle\}$;
    $label(s) := label(s) \cup \{\langle \varphi, \Omega_1 \cup \Omega_2, \mathcal{C}, O_1 \cup O_2 \rangle\}$;

Validating SCTP Simultaneous Open Procedure

Somsak Vanit-Anunchai*

School of Telecommunication Engineering, Institute of Engineering,
Suranaree University of Technology, Muang,
Nakhon Ratchasima, Thailand
somsav@sut.ac.th

Abstract. The Stream Control Transmission Protocol (SCTP) is a reliable unicast transport protocol originally specified by the Internet Engineering Task Force (IETF) in RFC 2960. After years of implementing and testing, defects and errors in RFC 2960 were reported and later fixed in RFC 4460. Incorporating those suggested fixes, IETF revised the SCTP specification and published RFC 4960, which replaces RFC 2960. Despite of being the revised specification, the descriptions of the simultaneous open and the restart procedures are still unclear and difficult to understand. To clarify this informal specification and gain insights, we formally model and analyse the association management using Coloured Petri Nets. In particular this paper focuses on the Tie-Tag operation and the simultaneous open procedure operating over the simplest channels, First In First Out (FIFO) with no loss. Our analysis reveals errors in which both sides are in ESTABLISHED but the verification tags in both Transmission Control Blocks do not match.

Keywords: Coloured Petri Nets, Procedure-based, Verification Tags, Tie-Tags, COOKIE ECHO

1 Introduction

The Stream Control Transmission Protocol (SCTP), RFC 4960 [8] is a unicast connection oriented transport protocol providing an error-free reliable flow of data between a client and a server. Originally it was designed by the Signalling Transport working group for transporting telephony signalling messages over UDP. Foreseeing its significance and great potential to become a major transport protocol, IETF decided to operate SCTP over IP instead.

After several years of implementation and testing, fifty-two defects in the original specification (RFC 2960) and solutions were discussed in RFC 4460 [9]. The IETF has published a revised version of the SCTP informal specification, RFC4960 [8], in September 2007, and RFC 2960 has become obsolete. Despite many years of implementing and testing SCTP, it is still important to have a proper formal model and to perform formal analysis of SCTP association management, especially when SCTP is designed for reliable data transfer

* Supported by the the Thai Network Information Center Foundation and the Thailand Research Fund Contract no.TRG 5380023.

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 233–249, 2013.
DOI: 10.1007/978-3-642-40213-5_15,
© IFIP International Federation for Information Processing 2013

such as signalling in Public Switching Telephone Networks. Previously in [10] we focused on modelling the typical procedure of SCTP association management. This paper places emphasis on the exception handling procedure (handling an unexpected packet) which is more complex. Analysis of a formal model in [10] illustrated that SCTP simultaneous open procedure in RFC 4960 could fall into an undesired final state in which both sides are in ESTABLISHED with mismatched VTAGs. However, it is arguable that these errors could happen only if a packet is reordered and delayed too long so that these defects are unlikely to occur. This paper discovers an error when SCTP operates over First in First out (FIFO) channel without losses. Thus, this error is more likely to occur than those errors previously found.

This paper is organised as follows. Section 2 provides an overview of SCTP association set up. Section 3 discusses the related work and contributions. A brief description of the CPN model of SCTP association management is given in Section 4. Section 5 presents the analysis results and a discussion of terminal markings. Section 6 presents conclusions and future work.

2 Overview of Stream Control Transmission Protocol

2.1 SCTP Packet Format

Figure 1(a) shows an SCTP packet comprising a common header and one or more chunks. The SCTP header contains 16 bit source and destination port numbers, a 32 bit verification tag (VTAG) and a 32 bit checksum. The VTAG is used to protect an association from blind attacks. Each end point keeps two values of VTAG: "local VTAG" and "peer's VTAG". "local VTAG" sometimes is called "My VTAG". In general, any received packets containing a VTAG differing from "local VTAG" will be discarded. On the other hand, sent packets will carry a VTAG equal to "peer's VTAG". These tag values are randomly selected at initialization and exchanged between the end points during association set up.

A Chunk is an information unit. According to RFC 4960, there are 12 different control chunks but only one data chunk. The control chunks are Init[1], InitAck, SACK, Heartbeat, HeartbeatAck, Abort, Shutdown, ShutdownAck, Error, CookieEcho, CookieAck and ShutdownComplete. Control chunks are used to setup and shutdown the association, selectively acknowledge, report error messages, monitor reachability of the peer, etc. Association setup uses a four-way handshake comprising four control chunks: Init; InitAck; CookieEcho and CookieAck. Graceful closing uses a three-way handshakes comprising three control chunks: the Shutdown; ShutdownAck and ShutdownComplete chunks. The Data transfer phase involves Data and SACK (Selective Acknowledgement) chunks. Further detail of the structure of chunks can be found in [8].

[1] Chunk names in the RFC are shown in all uppercase letters. To increase readability and distinguish them from SCTP States, the chunk names in this paper are given with only the first letters capitalized instead.

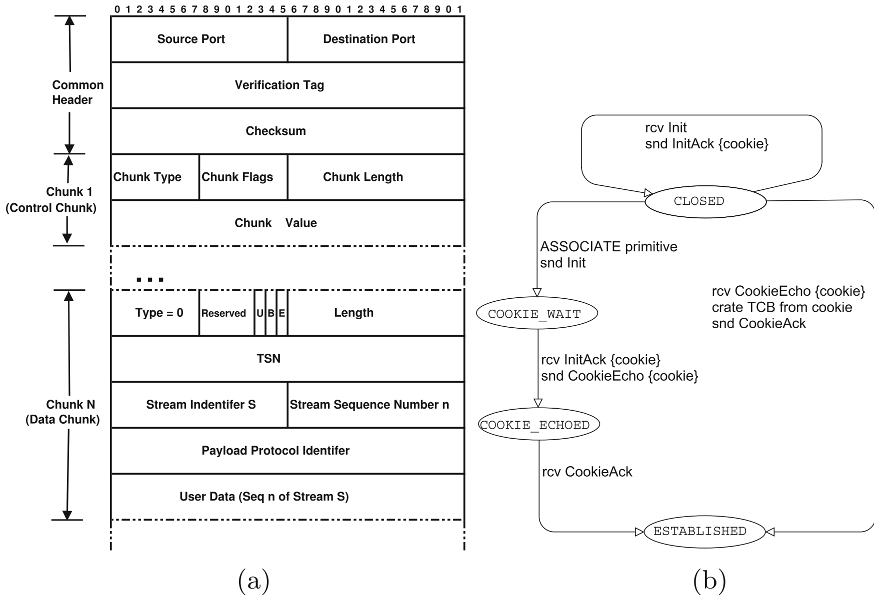


Fig. 1. (a) SCTP packet format. (b) SCTP state diagram: association set up (redrawn from [8])

2.2 SCTP Association Establishment Procedure

Normal Association Establishment Figure 1(b) shows the state diagram when SCTP sets up the association. Figure 2 shows a typical procedure of association establishment. An association between two nodes, A and Z, is initiated by an SCTP user on node "A" issuing an "ASSOCIATE" command. After receiving the "ASSOCIATE" primitive, node A sends an SCTP packet with VTAG equal to zero. This SCTP packet contains only an Init chunk with an initial tag to specify the VTAG of returning packets. Then node A enters the COOKIE_WAIT state. On receiving the Init chunk, node Z replies with an InitAck chunk indicating that it is willing to communicate with node A. The response includes node Z's initial tag number and encrypted cookie containing enough information to create node Z's Transmission Control Block (TCB). To prevent state exhaustion attacks node Z is still in CLOSED after replying with an InitAck. To acknowledge the InitAck, node A returns the cookie in a CookieEcho chunk and enters COOKIE_ECHOED. When carrying an Init or InitAck chunk, the SCTP packet comprises only one chunk. When sending a CookieEcho chunk, the SCTP packet may enclose Data chunks after the CookieEcho chunk. On receiving a CookieEcho from node A, node Z creates its TCB from the received cookie, enters the ESTABLISHED state, replies with CookieAck and is ready for data transfer. After receiving CookieAck, node A enters ESTABLISHED indicating that the association is established. During data transfer, endpoint nodes A and Z may exchange Data and SACK chunks.

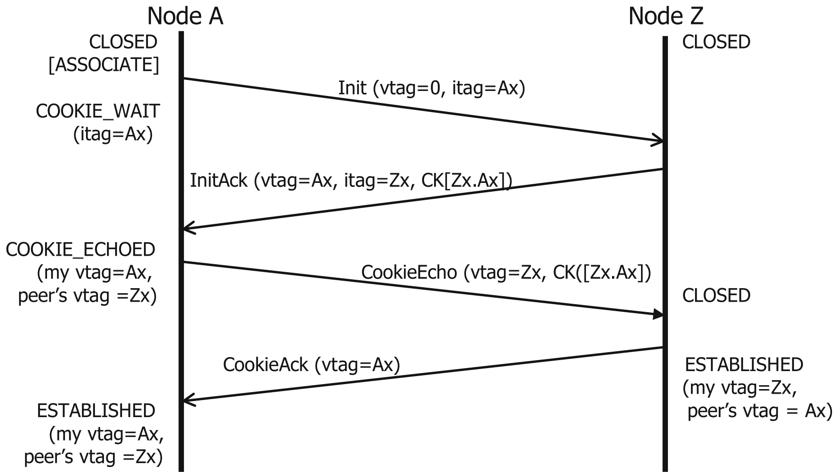


Fig. 2. Typical message sequence charts association set up

Handling Unexpected Init Chunks The rules for handling duplicate or unexpected Init, InitAck, CookieEcho, and CookieAck chunks are specified in the Section 5.2 of RFC 4960. When SCTP receives an unexpected Init before the association established, SCTP composes a state Cookie using its local VTAG and the initial tag found in the unexpected Init. The Cookie is attached to the outbound InitAck. When an unexpected Init is received after the association established, SCTP composes a state Cookie using a new random number for local VTAG and the initial tag found in the unexpected Init. This implies that the Cookie contains state information of a new connection. When an unexpected Init is received in SHUTDOWN_ACK_SENT, SCTP replies with ShuntDownAck.

Handling Unexpected CookieEcho Chunks This procedure specified in RFC 4960 is unclear and subtle. With reference to Table 2 in RFC 4960 (Fig. 3), VTAGs and Tie-Tags are compared to identify which action SCTP shall take. If the conditions in Fig. 3 are not met, SCTP silently discards the received CookieEcho chunk. However Section 5.2 of RFC 2960 and of RFC 4960 define Tie-Tags differently.

According to RFC 2960, Tie-Tags are stored in a state Cookie. They are copies of VTAGs from the existing TCB (local VTAG and peer's VTAG). These Tie-Tags are created when SCTP creates InitAck. Using the Tie-Tags in the restart procedure (Section 5.2.4.1 of RFC 4960), a newly restarting association can be tied to the original association without shutting down and starting a new association. The first two columns of Fig. 3 compare a pair of VTAGs in Cookie with a pair of VTAGs in existing TCB. The third and fourth columns of Fig. 3 compare a pair of Tie-Tags in Cookie with a pair of VTAGs in existing TCB.[2]

[2] [10] uses this comparison which is incorrect.

Local Tag	Peer's Tag	Local-Tie-Tag	Peer's-Tie-Tag	Action/ Description
X	X	M	M	(A)
M	X	A	A	(B)
M	0	A	A	(B)
X	M	0	0	(C)
M	M	A	A	(D)

Table 2: Handling of a COOKIE ECHO when a TCB Exists

```
Legend:

   X - Tag does not match the existing TCB.
   M - Tag matches the existing TCB.
   0 - No Tie-Tag in cookie (unknown).
   A - All cases, i.e., M, X, or 0.

Note: For any case not shown in Table 2, the cookie should be
silently discarded.
```

Fig. 3. Table 2 in RFC 4960 from [8]

In order not to reveal the true VTAGs of the existing association, RFC 4960 defines Tie-Tags as two 32-bit random numbers or 64-bit nonce. They are stored in both the state cookie and TCB. When we consider the third and fourth columns of Fig. 3, it is incorrect to compare 64-bit nonce with VTAGs. We should compare a pair of Tie-Tags (64-bit nonce) in Cookie with a pair of Tie-Tags (64-bit nonce) in existing TCB instead.[3]

Each row in Fig. 3 identifies the SCTP's action as follows:

- Action A is the restart scenario when the other side crashes and starts up in CLOSED. SCTP shall continue the association by replacing the existing VTAGs with the ones in Cookie and sending a CookieAck.
- Action B is the simultaneous open scenario when both sides attempt to start an association at about the same time. SCTP shall enter the ESTABLISHED state, update its peers VTAG from Cookie. and then send a CookieAck.
- Action C is when the Cookie is so delayed that SCTP has already sent an Init, received an InitAck and then sent a CookieEcho. The delayed Cookie arrives after the CookieEcho is sent. In this case the delayed Cookie is silently discarded.
- Action D is when both local and peer's VTAG in both Cookie and TCB are matched, SCTP shall enter the ESTABLISHED state and reply with CookieAck.

One subtle ambiguity is the meaning of the zero values in Fig. 3. The values of Tie-Tags are set to zero indicating that no previous TCB existed. Action C

[3] This paper uses this comparison.

requires the conditions that the values of Tie-Tags in the received cookie are zero. The value of peer's tag in TCB can be zero or unknown only when SCTP endpoint is in the COOKIE_WAIT state. It implies that action B in the third row of Fig. 3 occurs when SCTP endpoint is in COOKIE_WAIT.

Handling Unexpected InitAck and CookieAck Chunks An unexpected InitAck is simply discarded if SCTP is not in COOKIE_WAIT. SCTP also discards the CookieAck if it is not in COOKIE_ECHO.

3 Related Work

3.1 Modelling Approach

Coloured Petri Nets [5] are well known for modelling and analysing concurrent and complex systems including validating various transport protocols such as Wireless Application Protocol (WAP) [3], TCP [4], and Datagram Congestion Control Protocol (DCCP) [11]. Our model has been created and maintained using CPN Tools [2] which is a software package for the creation, editing, simulation and state space analysis of CPNs. It supports the hierarchical construction of CPN models [5], using constructs called *substitution transitions*. These transitions hide the details of subnets and allow further nesting of substitution transitions. This technique allows a complex specification to be managed as a series of hierarchically related pages.

According to [1], the hierarchical structure of the CPN model can be classified into three modelling styles: state-based; event-processing and procedure-based. Similar to state tables, the state-based style groups actions in the same state into a CPN page. ITU-T often describes their narrative specification based on the state tables. This approach has the advantage of readability and unspecified actions can be easily discovered during model construction. But its disadvantage comes from redundant specification of the same actions that are common across different states. Hence the event-processing style folds the similar actions across different states into a transition. An example of specification that uses event-processing style is RFC 793 [7] Transmission Control Protocol (TCP). While the event processing style makes the CPN model smaller and easier to maintain, it has some drawbacks with respect to readability. Thus [1] proposed the procedure-based modelling style, which structures the CPN model according to the protocol's functionality. Actually suitability of the modelling style depends on how the narrative specification is written. As long as the model can be read and understood easily alongside the narrative specification, it is a good modelling style. We notice that IETF's transport protocol specifications (TCP, SCTP and DCCP) are more suitable to the procedure-based style. During modelling SCTP association management [10], we discovered that this procedure-based style has two merits. First, using a state-based or event processing style a CPN page contains actions that are scattered in different sections of RFC 4960. Illustrated in [10], with the procedure-based style actions in each CPN page are confined to

only a few sections in RFC 4960. Our SCTP-CPN model in [10] is easy to read alongside RFC 4960. Second, the procedure-based CPN model comprises typical procedures (straight forward) and unexpected procedures (complex). Beginners can pay attention to the typical scenarios before getting into the complex procedures later.

3.2 Comparing to the SCTP-CPN Model by Others

Despite a lot of work on SCTP's security, performance and multi-homing, we have found only three works [12,6,10] that use Coloured Petri Nets to model SCTP association management operating over reordering channels. The CPN model in [6] followed the state-based style whereas [12] used the event-processing style similar to [4]. Our CPN model in [10] was the procedure-based style following the approach proposed in [1]. The work in [10] attempted to build the CPN models according to the revised specification, RFC 4960 while [6,12] used RFC 2960 which was obsolete. The CPN models in [6,12] were incomplete because [6] did not include the procedure when SCTP nodes receive duplicated or unexpected messages and [12] assumed no retransmission.

3.3 Contributions

The contribution of this paper is three-fold. First, while [10] illustrates a CPN model of typical SCTP's association management, it leaves out the model of handling an unexpected CookieEcho chunk partly because it was not well understood at that time. In this paper, we attempt to finish up the CPN model of handing an unexpected CookieEcho chunk. Second, [10] followed Fig. 5 (a restart example) of RFC 4960 and used the Tie-Tags as "old VTAGs" instead of 64-bit nonce. We compared the pair of Tie-Tags in Cookie with the pair of VTAG in existing TCB. Unfortunately, it turns out that Fig. 5 of RFC 4960 is incorrect[4]. Nevertheless, those errors uncovered in [10] were not related to this mistake. This paper attempts to rectify the mistake in [10]. We use Tie-Tags as 64-bit nonce and compare the pair of Tie-Tags in Cookie with the pair of Tie-Tags in existing TCB. This correction leads us to an undesired terminal marking in which both sides are in ESTABLISHED with a mismatched VTAG. This error scenario happens even when SCTP operates over FIFO channels without loss.

Third, after we have investigated the actions in Fig. 3 using state space analysis, we found two implementation flaws. Firstly, the implementor does not need to check the condition for action C because the Cookie will be discarded anyway if the conditions in Fig. 3 are not met. Secondly, the condition of action B in the third row (Fig. 3) has never been reached because the condition of action B in the second row is always satisfied before reaching the third row. We suggest the implementor checking the condition of the third row before checking the second row.

[4] See Transport Area Discussion Archive http://www.ietf.org/mail-archive/web/tsvwg/current/msg08603.html.

4 CPN Model of SCTP Association Management

Space limitation prevents us from including all CPN model pages. This paper focuses on handling an unexpected CookieEcho which is excluded from [10]. However, for sake of completeness, we shall briefly describe the model starting from the top level toward the `Unexpected CookieEcho and CookieAck` Page. The rest of the model and its declarations can be found in [10].

The top-level page of the SCTP-CPN model is illustrated in Fig. 4. Two substitution transitions (`SCTP_A` and `SCTP_Z`) represent the SCTP end point nodes, A and Z. Each side connects to five places. Places `User_A` and `User_Z` represent application users represented by `COMMAND`. Places `ITAG_A` and `ITAG_Z` contain 32-bit random values of the initial verification tags. Places `TCB_A` and `TCB_Z` model Transmission Control Block represented by `TCB`. Both end points are connected via two channel places, `CH_A2Z` and `CH_Z2A`. We assume that during association set up and closing down a packet contains only one chunk represented by `CHUNK`. To form a FIFO queue the channel places are represented by a list of `CHUNK` (`L_CHUNK`). The layout of the top level CPN page also reflects the well-known model of the n-layer in a layered protocol architecture. The application layer is placed on the top while the underlying medium layer is below the protocol entity. The substitution transitions, `SCTP_A` and `SCTP_Z` in Fig. 4 are linked to the second level page named `SCTP_Procedures` (Fig. 5 (a)). We divide `SCTP_Procedures` into five categories: normal events; unexpected events; retransmission; abort; and checking invalid tags. Substitution transition `UnexpectedEvents` in Fig. 5 (a) links to the CPN page `Unexpected` (Fig. 5 (b)). Handling unexpected receptions of SCTP control chunks is modelled by three CPN pages: `Int_IntAck`, `CookieEcho_CookieAck`, and `Shutdown`.

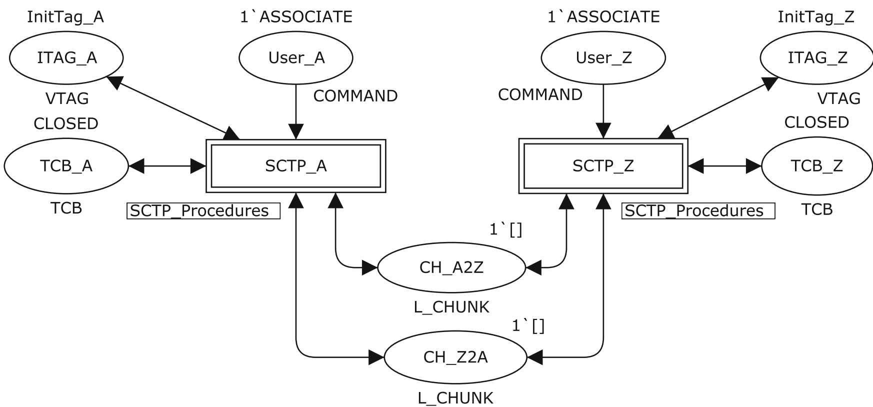


Fig. 4. The Top-level CPN page

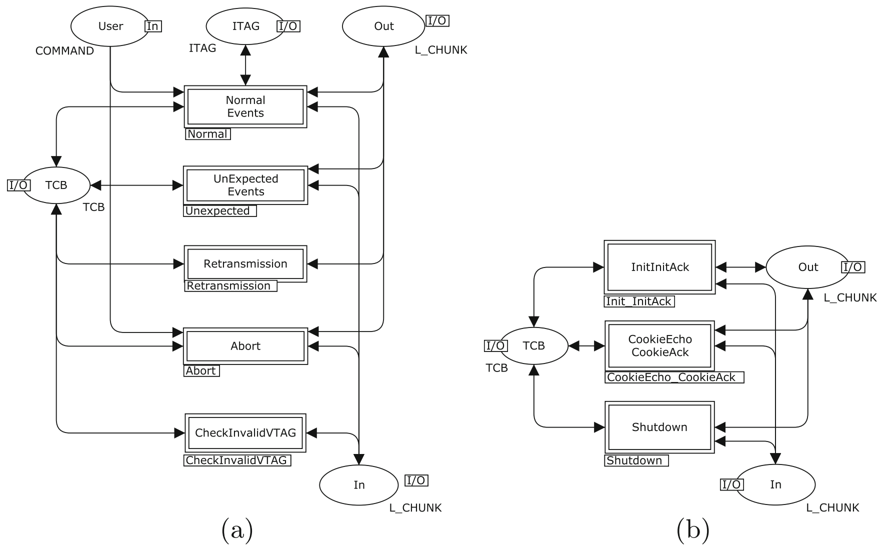


Fig. 5. (a) The SCTP_Procedures page. (b) The Unexpected page

4.1 Unexpected Init and InitAck Page

Figure 6 shows the CPN page dealing with the unexpected events of receiving Init and InitAck chunks in states other than CLOSED. Transitions `RcvInit_CK_WAIT` and `RcvInit_CK_ECHOED` model the actions according to Section 5.2.1 of RFC 4960 [8] when an endpoint receives an Init chunk in the COOKIE_WAIT or COOKIE_ECHOED state. The difference between these actions is that the Tie-Tags from the COOKIE_WAIT state are set to zeros but from COOKIE_ECHOED, they are set to 64-bit nonce. Transitions `Rcv_InitOtherThan` models the action according to Section 5.2.2 of RFC 4960 when the endpoints receive unexpected Init chunks in states other than CLOSED, COOKIE_WAIT, COOKIE_ECHOED and SHUTDOWN_ACK_SENT. The action is similar to that of transition `RcvInit_CK_ECHOED` but the "local VTAG" in the cookie and Initial Tag in the InitAck chunk are set to a new value instead of the old value of the Initial tag. Transition `RcvInit_in_SHUTDOWN_ACK_SENT` models the action according to the sixth paragraph of Section 9.2 of RFC 4960. After receiving an Init chunk in SHUTDOWN_ACK_SENT, the SCTP node discards the Init chunk but retransmits a ShutdownAck chunk. Transition `Rcv_InitAck` models the action according to Section 5.2.3 of RFC 4960. The SCTP node silently discards any unexpected InitAck chunks if receiving them in states other than COOKIE_WAIT.

4.2 Unexpected CookieEcho and CookieAck Page

Substitution transition `CookieEcho_CookieAck` in Fig. 5 (b) links to the CPN page `CookieEcho_CookieAck` (Fig. 7). The first four substitute transitions in

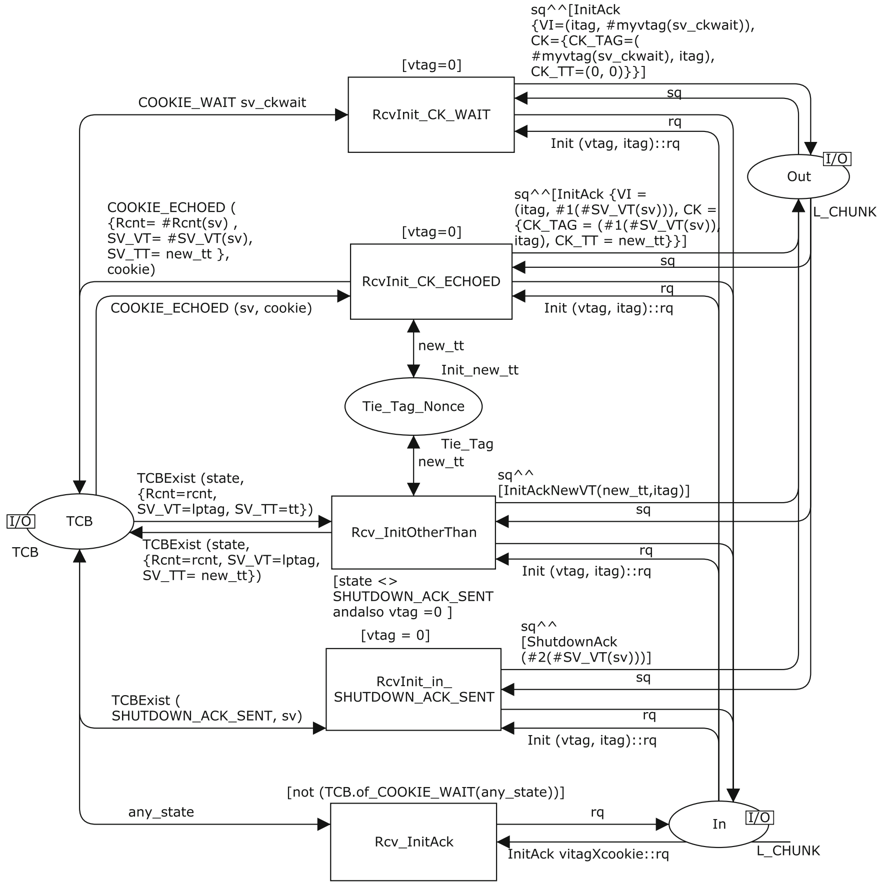


Fig. 6. The Unexpected'Init_InitAck page

Fig. 7 represent the actions when SCTP receives an unexpected CookieEcho chunk. The last transition models when SCTP receives CookieAck in states other than COOKIE_ECHOED.

The Restart page Substitute transition `Restart` in Fig. 7 is linked to the `Restart` page shown in Fig. 8. This page models action A of Fig. 3. SCTP replaces its VTAGs with the VTAGs in the received Cookie and replies with CookieAck. If SCTP is in SHUTDOWN_ACK_SENT, it retransmits ShutdownAck.

The Simultaneous_Open page Substitute transition `Simultaneous_Open` in Fig. 7 is linked to the `Simultaneous_Open` page shown in Fig. 9. This page models action B of Fig. 3. This page includes the actions when SCTP receives

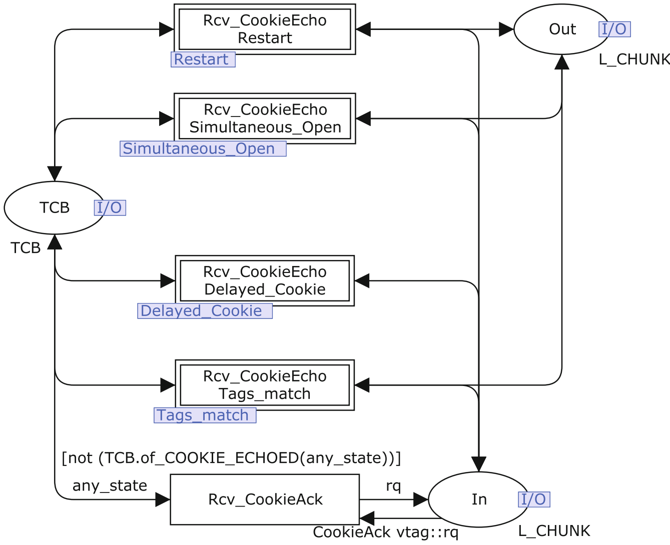


Fig. 7. The Unexpected'CookieEcho_CookieAck page

an unexpect CookieEcho in COOKIE_WAIT. It also include the actions when the conditions in Fig. 3 are not met (Case E).

The Delayed_Cookie page Substitute transition `Delayed_Cookie` in Fig. 7 is linked to the `Delayed_Cookie` page shown in Fig. 10. This page models action C of Fig. 3. SCTP silently discarded the delayed Cookie.

The Tags_match page Substitute transition `Tags_match` in Fig. 7 is linked to the `Tags_match` page shown in Fig. 11. This page models action D of Fig. 3. SCTP replies with CookieAck and enters ESTABLISHED.

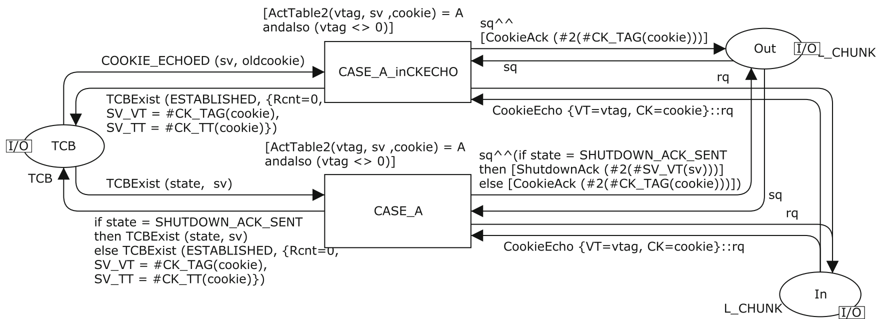


Fig. 8. The Restart page

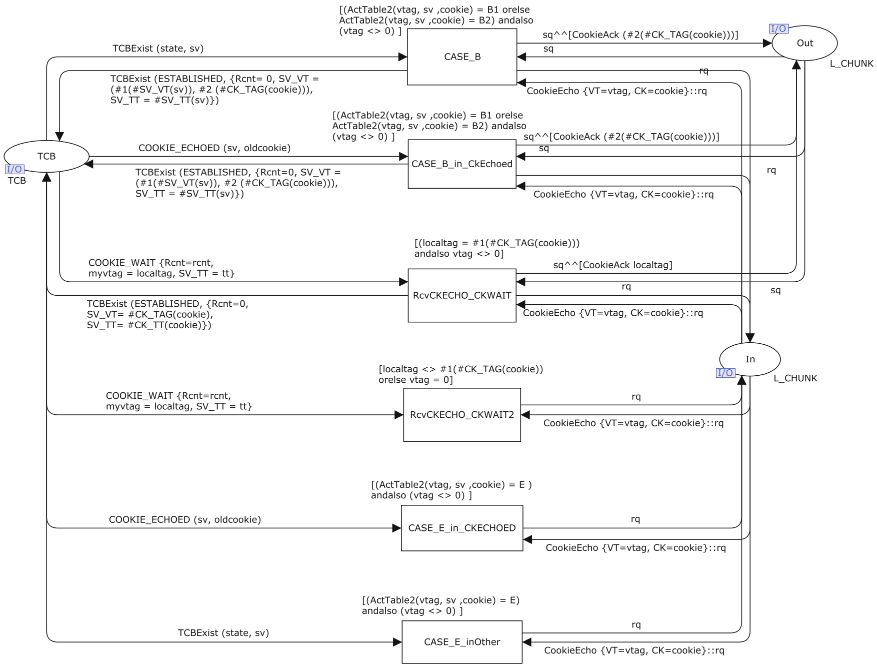


Fig. 9. The Simultaneous_Open page

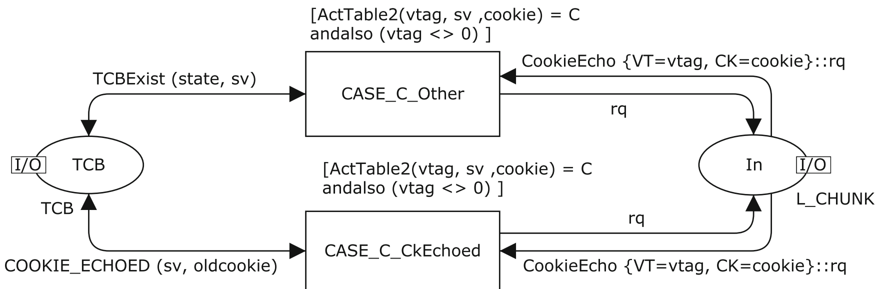


Fig. 10. The Delayed_Cookie page

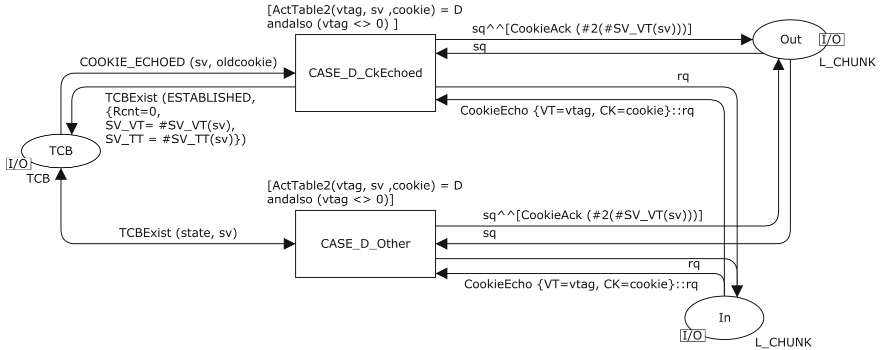


Fig. 11. The Tags_Matched page

5 Analysis of SCTP-CPN Association Management Model

5.1 Initial configuration

We analyse our SCTP association management model using CPN Tools version 3.2.2 on an Intel(R)Core i5 2.67 GHz computer with 4GB RAM. The SCTP-CPN model is initialised by distributing initial tokens to the places shown in Fig. 4. No packet is in both channel places. Places ITAG_A and ITAG_Z store initial verification tags which are randomly generated. Place Tie_Tag_Nonce in Fig. 6 contains a pair of 32-bit random numbers for Tie-Tags.

5.2 Analysis Results

The analysis results of our SCTP simultaneous open CPN model are shown in Table 1. The 2-tuple in the first column is the maximum retransmissions allowed for Init and CookieEcho. The state space tool in CPN Tools provides the number of nodes, arcs and terminal markings. In all cases the number of nodes and arcs in the Strongly Connected Component (SCC) Graph are the same as the number of nodes and arcs in the state space. Thus, no livelocks are found. We classify the terminal markings into four categories based on the SCTP endpoint states.

TYPE-I terminal marking (CLOSED-CLOSED) is a desirable terminal marking when the association cannot be established thus both sides go to CLOSED state (No connection). TYPE-III and TYPE-IV terminal markings[5] occur when one side is in ESTABLISHED while the other is in CLOSED. This can happen when the maximum number of retransmissions of the CookieEcho chunk is reached and the node enters CLOSED before CookieAck arrives. An example of this scenario is shown in Fig. 12. Although TYPE-III and TYPE-IV are unwanted, they are not harmful. This is because SCTP in CLOSED will report

[5] TYPE-III: node A terminates in CLOSED but node Z in ESTABLISHED.
TYPE-IV: node A terminates in ESTABLISHED but node Z in CLOSED.

Table 1. State space analysis results

Case	Nodes	Arcs	Time	Terminal Markings			
			(sec)	(I)CL-CL	(II)EST-EST	(III)CL-EST	(IV)EST-CL
(0,0)	278	444	-	1	15(0)	8	8
(0,1)	663	1,158	-	1	19(2)	9	9
(0,2)	1,353	2,554	00:00:02	1	21(2)	10	10
(0,3)	2,458	4,892	00:00:04	1	23(2)	11	11
(0,4)	4,098	8,458	00:00:06	1	25(2)	12	12
(0,5)	6,405	13,564	00:00:15	1	27(2)	13	13
(0,6)	9,523	20,548	00:00:25	1	29(2)	14	14
(1,0)	10,242	18,820	00:00:28	1	74(0)	37	37
(0,7)	13,608	29,774	00:00:22	1	31(2)	15	15
(0,8)	18,828	41,632	00:00:40	1	33(2)	16	16
(1,1)	27,433	54,104	00:02:51	1	102(8)	47	47
(1,2)	65,589	135,134	00:17:13	1	122(8)	57	57
(1,3)	139,919	296,178	01:29:02	1	142(8)	67	67

the failure to its user so that the user may decide to re-initiate the ASSOCIATE command. Thus the association can be restored as described in Fig. 5 of RFC 4960 (a restart example).

TYPE-II terminal markings should be desirable when both sides successfully establish the association. However when we check the verification tags stored in the TCBs, some terminal markings of TYPE-II are undesirable. “peer’s VTAG” of node A must equal “local VTAG” of node Z and vice versa, otherwise received data packets will be discarded. In column TYPE-II, the number in parenthesis is the number of TYPE-II terminal markings in which verification tags between both TCBs do not match each other.

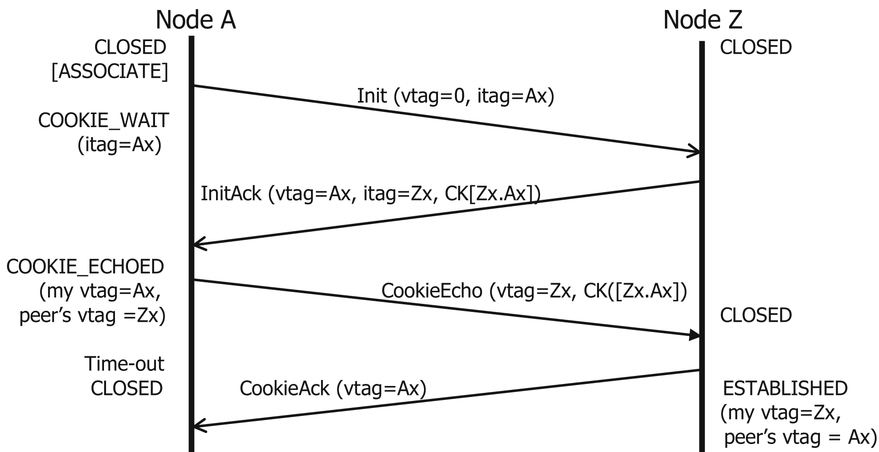


Fig. 12. A scenario leads to a terminal marking TYPE III (half open state)

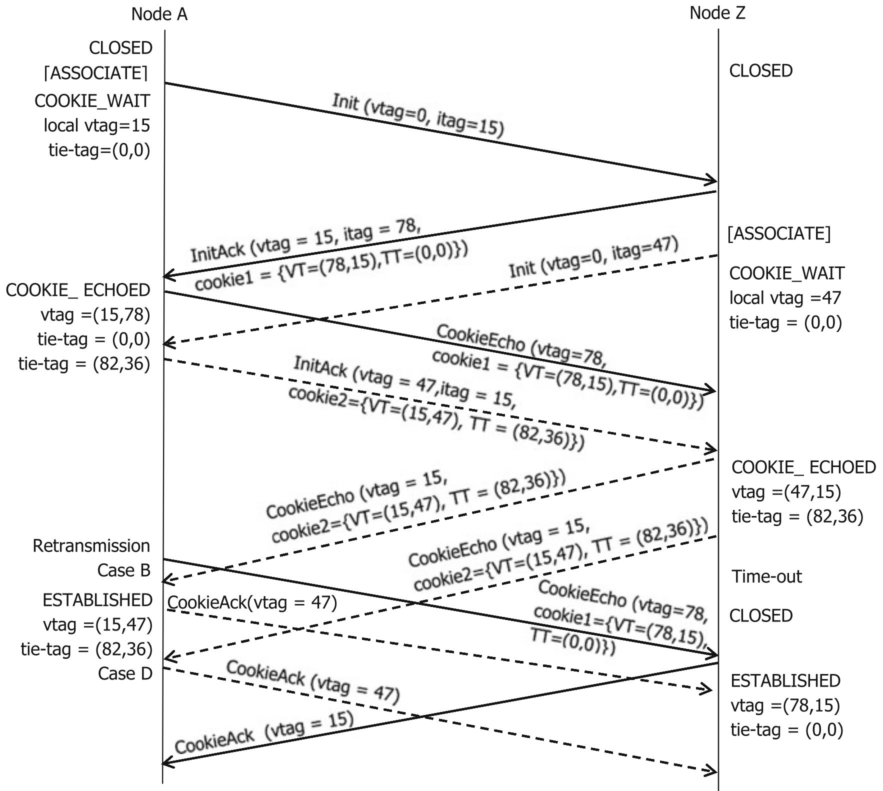


Fig. 13. A scenario leading to an undesired terminal marking TYPE II with mismatched VTAGs

Figure 13 shows a message sequence diagram leading to an undesired deadlock for case (0,1). Node A starts initiating the first connection (solid line). After replying with InitAck (itag=78), node Z initiates the second connection (dot line) using a different initial tag (itag=47). Node Z in COOKIE_WAIT keeps discarding the returned CookieEcho of the first connection (solid line) because the conditions in Fig. 3 are not met. After receiving InitAck of the second connection (dot line) and replying with CookieEcho, node Z stays in COOKIE_ECHOED state. Owing to the condition of action C in Fig. 3, node Z in COOKIE_ECHOED keeps discarding the returned CookieEcho of the first connection (solid line). When the timer expires or an intermittent fault occurs, node Z enters the CLOSED state. After node Z in CLOSED receives the CookieEcho of the first connection (solid line), it restores ESTABLISHED state from the received cookie. After node A in COOKIE_ECHOED, receives the CookieEcho of the second connection (dot line), it enters the ESTABLISHED state (*Action B*). Note that in Fig. 13 the CookieEcho of the first connection (solid line) is always sent by node A and the CookieEcho of the second connection (dot

line) is always sent by node Z. After both sides are in ESTABLISHED, "peer's VTAG" of node A (47) is not equal to "local VTAG" of node Z (78). Node A can receive data but will not receive any acknowledgement from node Z. Node Z cannot receive any data from node A. Node Z cannot get into the restart procedure immediately because it is not in CLOSED yet. Node Z and node A have to wait for time-out before closing down the association.

6 Conclusions and Future Work

This paper has presented a Coloured Petri Nets model and analysis of SCTP simultaneous open procedure. While constructing the SCTP-CPN model, we identify the incorrect description of the *Tie-Tags* in RFC 4960. Our rigorous analysis shows that SCTP simultaneous open procedure, operating over FIFO channels with no loss, could fail into an undesired deadlock. Both sides in ESTABLISHED could have mismatched verification tags in their TCBs. When the server is located behind middle-boxes such as fire wall or Network Address Translators (NAT), the transport protocols (UDP, TCP, DCCP and SCTP) normally use simultaneous open procedures. Nowadays NATs are widely deployed so that these defects in simultaneous open procedures should not be overlooked.

Formal analysing connection management of the other transport protocols: WAP [3], TCP [4] and DCCP [11], reveal no error when the protocols operate over FIFO channels with no loss. Usually errors could appear when the protocols operate over reordering and/or lossy channels. But the deadlock shown in Fig. 13 does not require any reordered or irregularly delayed packets. Although the odds of this particular scenario is low, the number of terminal markings Type II in Table 1 suggests that depending on the number of retransmitted Init Chunks, there are a large number of possible scenarios leading to the similar deadlock.

SCTP includes various capabilities, such as the restart and multi-homing procedures, aiming for high reliability or fault tolerance applications. When SCTP nodes enter the deadlock state, they have to wait for time-out before closing down the association. This delay degrades SCTP performance. As far as we are aware, this kind of errors has not been raised before. Given the above reasons and the enormous number of potential SCTP connections in the Internet, we consider that this problem could be a serious threat to SCTP applications especially when the high reliability is required.

In future, we are interested in modelling SCTP operating via Network Address Translators (NAT) with multi-homing functions.

Acknowledgements. This work is supported by Research Grant from the Thai Network Information Center Foundation and the Thailand Research Fund. The author is thankful to Jonathan Billington, Guy Gallasch and the anonymous reviewers. Their constructive feedback has helped the author improve the quality of this paper.

References

1. Billington, J., Vanit-Anunchai, S.: Coloured Petri Net Modelling of an Evolving Internet Standard: the Datagram Congestion Control Protocol. Fundamenta Informaticae 88(3), 357–385 (2008)
2. CPN Tools home page, http://wiki.daimi.au.dk/cpntools-help/_home.wiki
3. Gordon, S.: Verification of the WAP Transaction Layer uisng Coloured Petri Nets. PhD thesis, Institute for Telecommunications Research and Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia (November 2001)
4. Han, B.: Formal Specification of the TCP Service and Verification of TCP Connection Management. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia (December 2004)
5. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
6. Martins, M.G.M.: Modelagem e Análise Formal de algumas Funcionalidades de um Protocolo de Transporte Atrvés das Redes de Petri. Master's thesis, Instituto Nacional de Telecomunicações (INATEL), Santa Rita do Sapucaí, Brazil (December 2003) (Only available in Portuguese)
7. Postel, J.: Transmission Control Protocol (TCP), RFC793 (September 1981), http://www.rfc-editor.org/rfc/rfc793.txt
8. Stewart, R. (ed.): Stream Control Transmission Protocol (SCTP), RFC4960 (September 2007)
9. Stewart, R., Arias-Rodriguez, I., Poon, K., Caro, A., Tuexen, M.: Stream Control Transmission Protocol (SCTP) Specification Errata and, Issues, RFC4460 (September 2007)
10. Vanit-Anunchai, S.: Toward Formal Modelling and Analysis of SCTP Connection Management. In: The 9th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Department of Computer Science, University of Aarhus, October 20–23, pp. 163–182 (2008)
11. Vanit-Anunchai, S.: An Investigation of the Datagram Congestion Control Protocol's Connection Management and Synchronisation Procedures. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia (November 2007)
12. Wang, J., Zhang, S., Chen, F.: Modelling and Verification of SCTP Association Management Based on Coloured Petri Nets. In: 2008 ISECS International Colloquium on Computing, Communication, Control, and Management, Guangzhou, China, August 3–4, pp. 379–383. IEEE Computer Society (2008)

Improving Time Bounded Reachability Computations in Interactive Markov Chains*

Hassan Hatefi[1,2] and Holger Hermanns[1]

[1] Saarland University – Computer Science, Saarbrücken, Germany
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
hhatefi@depend.cs.uni-saarland.de, hermanns@cs.uni-saarland.de

Abstract. Interactive Markov Chains (IMCs) are compositional behaviour models extending both Continuous Time Markov Chain (CTMC) and Labeled Transition System (LTS). They are used as semantic models in different engineering contexts ranging from ultramodern satellite designs to industrial system-on-chip manufacturing. Different approximation algorithms have been proposed for model checking of IMC, with time bounded reachability probabilities playing a pivotal role. This paper addresses the accuracy and efficiency of approximating time bounded reachability probabilities in IMC, improving over the state-of-the-art in both efficiency of computation and tightness of approximation. Experimental evidence is provided by applying the new method on a case study.

1 Introduction

Why IMCs? Over the last decade, a formal approach to quantitative performance and dependability evaluation of concurrent systems has gained maturity. At its root are continuous-time Markov chains for which efficient and quantifiably precise solution methods exist [3]. On the specification side, continuous stochastic logic (CSL) [1,3] enables the specification of a large spectrum of performance and dependability measures. A CTMC can be viewed as a labelled transition system (LTS) whose transitions are delayed according to exponential distributions. Opposed to classical concurrency theory models, CTMCs neither support compositional modelling [23] nor do they allow nondeterminism in the model. Among several formalisms that overcome these limitations [7,21,24,25], interactive Markov chains (IMCs) [22] stand out. IMCs conservatively extend classical concurrency theory with exponentially distributed delays, and this induces several further benefits [8]. In particular, it enables compositional modelling with intermittent weak bisimulation minimisation [21] and allows to augment existing untimed process algebra specifications with random timing [7]. Moreover, the IMC formalism is not restricted to exponential delays but allows to encode

* This work has been supported by the DFG as part of SFB/TR 14 AVACS, by the DFG/NWO bilateral project ROCKS, and by the European Union FP7-ICT projects MEALS, grant agreement no. 295261, and SENSATION, grant agreement no. 318490.

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 250–266, 2013.
DOI: 10.1007/978-3-642-40213-5_16,
© IFIP International Federation for Information Processing 2013

arbitrary phase-type distributions such as hyper- and hypoexponentials [28]. Since IMCs smoothly extend classical LTSs, the model has received attention in academic as well as in industrial settings [6,13,12,16].

Why time bounded reachability? The principles of model checking IMCs are by now well understood. One analysis strand, implemented for instance in CADP [17], resorts to CSL model checking of CTMCs. But this is only applicable if the weak bisimulation quotient of the model is indeed a CTMC, which cannot be always guaranteed. This is therefore a partial solution technique, albeit it integrates well with compositional construction and minimisation approaches, and is the one used in industrial applications. The approximate CSL model checking problem for IMCs has been solved by Neuhäusser and Zhang [26,29]. Most of the solution resorts to untimed model checking [5]. The core innovation lies in the solution of the time bounded model checking problem, that is needed to quantify a *bounded until formula* subject to a (real-valued) time interval. The problem is solved by splitting the time interval into equally sized digitisation steps, each small enough such that with high probability at most one Markov transition occurs in any step.

However, the practical efficiency and accuracy of this approach to evaluate time bounded reachability probabilities turns out substantially inferior to the one known for CTMCs, and this limits applicability to real industrial cases. As a consequence, model checking algorithms for other, less precise, but still highly relevant properties have been coined [19], including expected reachability and long run average properties.

Our contribution. We revisit the approximation of time bounded reachability probabilities so as to arrive at an improved computational approach. For this, we generalise the digitisation approach of Neuhäusser and Zhang [26,29] by considering the effect of multiple Markov transition firings in a time interval of length δ. We show that this can be exploited by a tighter error bound, and thus a more accurate computation. We put the theoretical improvement into practice by proposing a new algorithm to solve time bounded reachability in IMCs. Empirical results demonstrate that the improved algorithm can gain more than one order of magnitude speedups.

2 Interactive Markov Chain

An Interactive Markov Chain (IMC) is a model that generalises both CTMC and LTS. In this section, we provide the definition of IMC and the necessary concepts relating to it.

Definition 1. (IMC) *An IMC [21] is a tuple $\mathcal{M} = (S, Act, \longrightarrow, \dashrightarrow, s_0)$, where*

- *S is a finite set of states,*
- *Act is a set of actions, including τ, representing the internal invisible action,*
- *$\longrightarrow \subset S \times Act \times S$ is a set of interactive transitions,*
- *$\dashrightarrow \subset S \times \mathbb{R}_{\geq 0} \times S$ is a set of Markov transitions,*
- *s_0 is the initial state.*

Maximum progress vs. urgency. States of an IMC are partitioned into *interactive*, *Markov* and *hybrid*. Interactive (Markov) states have only interactive (Markov) outgoing transitions, while hybrid states have transitions of both types. Let S_I, S_M and S_H be the set of interactive, Markov and hybrid states respectively. An IMC might have states without any outgoing transition. For the purpose of this paper, any such state is turned into a Markov state by adding a self loop with an arbitrary rate. We distinguish between closed and open IMCs. An open IMC can interact with the environment and in particular, can be composed with other IMCs, e.g. via parallel composition. For such models, a *maximum progress assumption* [21] is imposed which implies that τ-transitions take precedence over Markov transitions whenever both are enabled in a state. In contrast, a closed IMC is not subject to any further communication and composition. In this paper, we assume that the models we are going to analyse are closed, and impose the stronger *urgency assumption* which means that any interactive transition has precedence over Markov transitions, i.e. interactive transitions are taken immediately whenever enabled in a state, leaving no chance for enabled Markov transitions. Consequently, in a closed IMC, hybrid states can be regarded as interactive states.

Branching probabilities. A (probability) distribution μ over a discrete set S is a function $\mu : S \rightarrowtail [0,1]$ such that $\sum_{s \in S} \mu(s) = 1$. If $\mu(s) = 1$ for some $s \in S$, μ is a *Dirac* distribution denoted by μ_s. Let $Dist(S)$ be the set of all distributions over set S. For uniformity of notations, we use a distinguished action $\perp \notin Act$ to indicate Markov transitions and extend the set of actions to $Act_\perp = Act \uplus \{\perp\}$. For $s \in S$, we define $Act_\perp(s)$ as the set of *enabled actions* in state s. If s is a Markov state, $Act_\perp(s) = \{\perp\}$, otherwise $Act_\perp(s) = \{\alpha \mid (s, \alpha, s') \in \longrightarrow\}$. The rate between state s and s' is defined by $rate(s, s') = \sum_{(s,\lambda,s') \in \dashrightarrow} \lambda$. Then $E(s) = \sum_{s' \in S} rate(s, s')$ denotes the sum of outgoing rates of state s. Using these concepts, we define the *branching probability* matrix for both interactive and Markov states by

$$\mathbf{P}(s, \alpha, s') = \begin{cases} 1 & s \in S_I \wedge (s, \alpha, s') \in \longrightarrow \\ \frac{rate(s,s')}{E(s)} & s \in S_M \wedge \alpha = \perp \\ 0 & \text{otherwise} \end{cases}$$

Example 1. Let $\mathcal{M}$ be the IMC in Figure 1. s_1 and s_3 are Markov states, while s_2 is an interactive state. Initial state s_0 is a hybrid state, since it has both interactive and Markov transitions enabled. Considering $\mathcal{M}$ as a closed IMC, the urgency assumption allows us to ignore $(s_0, 0.5, s_2) \in \dashrightarrow$ and consider s_0 as an interactive state. Under this assumption, interactive transitions are instantaneously fired after zero time delay. Conversely, the sojourn time in a Markov state s is exponentially distributed with rate $E(s)$. For example, the probability to leave s_1 within δ time unit is $1 - e^{-5\delta}$ ($E(s_1) = 2 + 3 = 5$). At this point, branching probabilities determine the distribution of evolving to next states. For s_1, $\mathbf{P}(s_1, \perp, s_0) = \frac{2}{5}$ and $\mathbf{P}(s_1, \perp, s_3) = \frac{3}{5}$, as a result the probabilities to go to s_0 and s_3 after spending at most δ time unit in s_1 are $\frac{2}{5}(1 - e^{-5\delta})$ and $\frac{3}{5}(1 - e^{-5\delta})$ respectively.

Behavioural aspects. Like in other transition systems, an execution in an IMC is described by a path. Formally, a finite path is a finite sequence $\pi = s_0 \xmapsto{t_0,\alpha_0} s_1 \cdots s_{n-1} \xmapsto{t_{n-1},\alpha_{n-1}} s_n$ with $\alpha_i \in Act_\perp, t_i \in \mathbb{R}_{\geq 0}, i = 0 \cdots n-1$. We use $|\pi| = n$ as the length of π and $last(\pi) = s_n$ as the last state of π. Each step of a path π describes how the IMC evolves from one state to the next, with what action and after spending what state sojourn time. For example, when the IMC is in an interactive state $s \in S_I$, it must immediately (in zero time) choose some action $\alpha \in Act_\perp(s)$ and go to state s'. This gives rise to the finite path $s \xmapsto{0,\alpha} s'$. On the other hand, if $s \in S_M$, the IMC can stay for $t > 0$ time units and then choose the next state s' based on the distribution $\mathbf{P}(s, \perp, \cdot)$ by $s \xmapsto{t,\perp} s'$. An infinite path specifies an infinite execution of an IMC. We use $Paths^*$ and $Paths^\omega$ to denote the set of finite and infinite paths, respectively. By dropping the sojourn times from a path, we obtain the time-abstract path. We use subscript ta to refer to the set of time-abstract finite and infinite paths (i.e. $Paths^*_{ta}$ and $Paths^\omega_{ta}$).

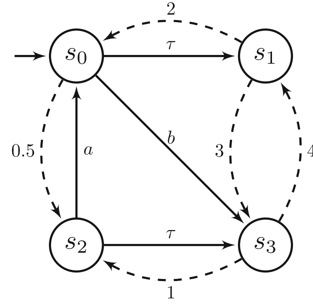


Fig. 1. An exemplary IMC

Resolving nondeterminism. In states with more than one interactive transitions, the resolution of the transition to take is nondeterministic, just as in the LTS setting. This nondeterminism is resolved by schedulers. The most general scheduler class maps a finite and possibly timed path to a distribution over the set of interactive transitions enabled in the last state of the path:

Definition 2. (Generic Scheduler) *A generic scheduler over IMC* $\mathcal{M} = (S, Act, \longrightarrow, \dashrightarrow, s_0)$ *is a function,* $A : Paths^* \rightarrowtail Dist(\longrightarrow)$*, where the support of* $A(\pi)$ *is a subset of* $(\{last(\pi)\} \times Act \times S) \cap \longrightarrow$ *and* $last(\pi) \in S_I$*.*

For a finite path π, a scheduler specifies how to resolve nondeterminism on the last state of π which is an interactive state. It gives a distribution over the set of enabled transitions of $last(\pi)$. We use the term *Gen* to refer to the set of all generic schedulers. Following the definition of schedules, the probability measure can be uniquely defined over the $\sigma-$algebra on $Paths^\omega$, given scheduler A and initial state s, denoted by $Pr^\omega_{A,s}$ [26].

Non-zenoness. Owing to the presence of immediate state changes, an IMC might exhibit *Zeno behaviour*, where infinitely many interactive transitions are taken in finite or zero time. This is an unrealistic phenomenon, characterised by an infinite path π, where the time spent on π does not diverge, called a Zeno path. To exclude such unrealistic phenomena, we restrict our attention to models where the probability of Zeno behaviour is zero. This means that $\forall A \in Gen,\ \forall s \in S.\ Pr^\omega_{A,s}(\Pi_{<\infty}) = 0$, where $\Pi_{<\infty}$ is the set of all Zeno paths.

This condition implies that starting from any interactive states, we must reach the set of Markov states with probability one. In the remainder of this paper, we therefore restrict to such models.

3 Time Bounded Reachability

CSL model checking of time bounded until properties plays a pivotal role in quantitative evaluation of IMCs. It can be reduced to time bounded reachability analysis, by a well-known technique [2] of making target states absorbing. This section reviews the current state-of-the-art [26,29] of solving time bounded reachability problems in IMC. Section 4 will discuss how can we improve upon that.

Fixed point characterisation. We first discuss the fixed point characterisation for the maximum probability to reach a set of goal states within an interval of time. For this, let $\mathcal{I}$ and $\mathcal{Q}$ be the set of all nonempty nonnegative real intervals with real and rational bounds respectively. For $I \in \mathcal{I}$ and $t \in \mathbb{R}_{\geq 0}$, we define $I \ominus t = \{x - t \mid x \in I \wedge x \geq t\}$. If $I \in \mathcal{Q}$ and $t \in \mathbb{Q}_{\geq 0}$, then $I \ominus t \in \mathcal{Q}$. Given IMC $\mathcal{M}$, a time interval $I \in \mathcal{I}$ and a set of goal states $G \subseteq S$, the set of all paths that reach the goal states within interval I is denoted by $\Diamond^I G$. Let $p_{\max}^{\mathcal{M}}(s, \Diamond^I G)$ be the maximum probability of reaching the goal states within interval I if starting in state s at time 0. In formal terms, it is the supremum ranging over all possible *Gen* schedulers, of the probability measures on the induced paths: $p_{\max}^{\mathcal{M}}(s, \Diamond^I G) = \sup_{A \in Gen} Pr_{A,s}^{\omega}(\Diamond^I G)$. The next lemma recalls a characterisation of $p_{\max}^{\mathcal{M}}(s, \Diamond^I G)$ as a fixed point. That of $p_{\min}^{\mathcal{M}}(s, \Diamond^I G)$ is dealt with similarly.

Lemma 1. (Fixed Point Characterisation for IMCs [26, Theorem 6.1]) *Let $\mathcal{M}$ be an IMC, $G \subseteq S$ be a set of goal states and $I \in \mathcal{I}$ with* $\inf I = a$ *and* $\sup I = b$. *$p_{\max}^{\mathcal{M}} : S \times \mathcal{I} \rightarrowtail [0,1]$ is the least fixed point of the higher-order operator $\Omega : (S \times \mathcal{I} \rightarrowtail [0,1]) \rightarrowtail (S \times \mathcal{I} \rightarrowtail [0,1])$, which is:*

1. *For $s \in S_M$*

$$\Omega(F)(s,I) = \begin{cases} \int_0^b E(s)e^{-E(s)t} \sum_{s' \in S} \mathbf{P}(s,\perp,s') F(s', I \ominus t)\, \mathrm{d}t & s \notin G \\ e^{-E(s)a} + \int_0^a E(s)e^{-E(s)t} \sum_{s' \in S} \mathbf{P}(s,\perp,s') & \\ \qquad \times F(s', I \ominus t)\, \mathrm{d}t & s \in G \end{cases}$$

2. *For $s \in S_I$*

$$\Omega(F)(s,I) = \begin{cases} 1 & s \in G \wedge 0 \in I \\ \max_{(s,\alpha,s') \in \longrightarrow} F(s', I) & \textit{otherwise} \end{cases}$$

Interactive Probabilistic Chain. The above characterisation provides an integral equation system of the maximum time interval bounded reachability probability. But this system is in general not directly tractable algorithmically [2]. To circumvent this problem, the fixed point characterisation can be approximated by a digitisation [26,29] approach. Intuitively, the time interval is split into equally sized subintervals, which we call digitisation steps. It is assumed that the digitisation constant δ is small enough such that with high probability it carries at most one Markov transition firing. This assumption reduces an IMC to an Interactive Probabilistic Chain (IPC) [12]. An IPC is a digitised version of IMC, obtained by summarising the behaviour of an IMC at equidistant time points.

Definition 3. *An IPC is a tuple $\mathcal{D} = (S, Act, \longrightarrow, \dashrightarrow_d, s_0)$, where S, Act, $\longrightarrow$ and s_0 are as Definition 1 and $\dashrightarrow_d \subset S \times Dist(S)$ is the set of digitised Markov transitions.*

A digitised Markov transition specifies with which probability a state evolves to its successors after taking one time step. The notion of digitised Markov transition resembles the one-step transition matrix in DTMC. The concepts of closed and open models carry over to IPC. As we do not have the notion of continuous time, paths in IPC can be seen as time-abstract paths in IMC, implicitly still counting digitisation steps, and thus discrete time.

Digitisation from IMC to IPC. We now recall the digitisation that turns an IMC into an IPC. Afterwards, we explain how reachability computation in an IMC can be approximated by analysis on IPC, for which there exists a proved error bound.

Definition 4. (Digitisation [26]) *Given IMC $\mathcal{M} = (S, Act, \longrightarrow, \dashrightarrow, s_0)$ and a digitisation constant δ, $\mathcal{M}_\delta = (S, Act, \longrightarrow, \dashrightarrow_\delta, s_0)$ is an IPC constructed from digitisation of $\mathcal{M}$ with respect to digitisation constant δ and $\dashrightarrow_\delta = \{(s, \mu^s) | s \in S_M\}$, where*

$$\mu^s(s') = \begin{cases} (1 - e^{-E(s)\delta})\mathbf{P}(s, \perp, s') & s' \neq s \\ (1 - e^{-E(s)\delta})\mathbf{P}(s, \perp, s') + e^{-E(s)\delta} & s' = s \end{cases}$$

The digitisation in Definition 4 approximates the original model by assuming that at most one Markov transition in $\mathcal{M}$ can fire in each step of length δ. It is specified by distribution μ^s, which contains the probability of having either one or no Markov transition in $\mathcal{M}$ from state s within a time interval of length δ. Using the fixed point characterisation above, it is possible to relate reachability analysis in an IMC to reachability analysis in its associated IPC [26], together with an error bound. We recall the result here:

Theorem 1. (Error Bound [26]) *Given IMC $\mathcal{M} = (S, Act, \longrightarrow, \dashrightarrow, s_0)$, a set of goal states $G \subseteq S$, a time interval $I \in \mathcal{Q}$ such that $a = \inf I$ and $b = \sup I$ with $0 \leq a < b$. and $\lambda = \max_{s \in S_M} E(s)$. Assume digitisation step $\delta > 0$ is*

selected such that $b = k_b\delta$ *and* $a = k_a\delta$ *for some* $k_b, k_a \in \mathbb{N}$. *For all* $s \in S$ *it holds*

$$p_{\max}^{\mathcal{M}_\delta}(s, \diamondsuit^{(k_a,k_b]}G) - k_a\frac{(\lambda\delta)^2}{2} \leq p_{\max}^{\mathcal{M}}(s, \diamondsuit^I G) \leq p_{\max}^{\mathcal{M}_\delta}(s, \diamondsuit^{(k_a,k_b]}G) + k_b\frac{(\lambda\delta)^2}{2} + \lambda\delta$$

For the proof of Theorem 1 see [26, Theorem 6.5].

Time bounded computation in IPC. We briefly review the maximum time bounded reachability computation in IPC [29]. At its core, a modified *value iteration* algorithm is carried out. Given an IPC, a set of goal states and a step interval, the algorithm iteratively proceeds by taking two different phases. In the first phase, reachability probabilities starting from all interactive states are updated. This is done by selecting the maximum from reachability probabilities of Markov states that are reachable from each interactive state. The second phase updates the reachability probabilities from Markov states by taking a digitised time step. The algorithm iterates until the last digitised time step is processed. For more details about the algorithm we refer to [29].

4 Improving Time Bounded Reachability Computation

In this section, we generalise the previously discussed technique for computing maximum time bounded reachability. As before, we approximate the fixed point characterisation of IMC using a digitisation technique. However instead of considering at most one, we consider at most n Markov transition firing(s) in a digitisation step, for n being an arbitrary natural number. This enables us to establish a tighter error bound. Alternatively, an increased n lets us to choose a larger digitisation constant δ, without compromising the original error bound. A larger digitisation constant implies fewer iterations, thus speeding up the overall runtime of the algorithm.

Higher-order approximation. When developing an approximation of n-th order of the maximum reachability probability, we first restrict ourselves to intervals with zero lower bounds.

Definition 5. *Given IMC* $\mathcal{M} = (S, Act, \longrightarrow, \dashrightarrow, s_0)$, *a set of goal states* $G \subseteq S$, *an interval* $I \in \mathcal{Q}$ *such that* $\inf I = 0$ *and* $\sup I = b$. *Assume digitisation step* $\delta > 0$ *is selected such that* $b = k_b\delta$ *for some* $k_b \in \mathbb{N}$. *We define* $p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^I G) = 1$ *if* $s \in G$, *and for* $s \in S \setminus G$:

$$p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^I G) = \begin{cases} A_{I,n}^n(s, \delta) & s \in S_M \setminus G \\ \max_{(s,\alpha,s') \in \longrightarrow} p_{\max}^{\mathcal{M}_\delta(n)}(s', \diamondsuit^I G) & s \in S_I \setminus G \end{cases}$$

and for $0 \leq k \leq n$ *and* $0 \leq \Delta \leq \delta$:

$$A_{I,n}^k(s, \Delta) = \begin{cases} \int_0^\Delta E(s)e^{-E(s)t}\sum_{s' \in S}\mathbf{P}(s, \perp, s')A_{I,n}^{k-1}(s', \Delta \\ \qquad -t)\,\mathrm{d}t + e^{-E(s)\Delta}p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^{I \ominus \delta}G) & s \in S_M \setminus G \wedge k > 0 \\ p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^{I \ominus \delta}G) & s \in S_M \setminus G \wedge k = 0 \\ \max_{(s,\alpha,s') \in \longrightarrow} A_{I,n}^k(s, \Delta) & s \in S_I \setminus G \end{cases}$$

Intuitively $A^k_{I,n}(s, \Delta)$ is the maximum probability to reach G from state s inside $I \ominus (\delta - \Delta)$ by having up to k Markov transition(s) in the first Δ time unit and up to n Markov transition(s) in each digitisation step δ afterwards. This approximation represents the behaviour of the original model more faithfully, thus leading to a better error bound. Theorem 2 quantifies the quality of this approximation.

Theorem 2. *Given IMC $\mathcal{M} = (S, Act, \longrightarrow, \dashrightarrow, s_0)$, a set of goal states $G \subseteq S$, an interval $I \in \mathcal{Q}$ with $\inf I = 0$, $\sup I = b$ and $\lambda = \max_{s \in S_M} E(s)$. Assume digitisation step $\delta > 0$ is selected such that $b = k_b \delta$ for some $k_b \in \mathbb{N}$ and $n > 0$ is the order of approximation. For all $s \in S$ it holds*

$$p^{\mathcal{M}_\delta(n)}_{\max}(s, \Diamond^I G) \leq p^{\mathcal{M}}_{\max}(s, \Diamond^I G) \leq p^{\mathcal{M}_\delta(n)}_{\max}(s, \Diamond^I G) + 1 - e^{-\lambda b}\Big(\sum_{i=0}^{n} \frac{(\lambda\delta)^i}{i!}\Big)^{k_b}$$

The proof of Theorem 2 is tedious, basically following and generalising the proof of [26, Theorem 6.3]. We provide the proof for the case $n = 2$ in the appendix and discuss how it can be extended to the general case. The core insight is, intuitively speaking, as follows. We can view the error as the probability of more than n Markov transition(s) firing in at least one digitisation step. Due to independence of the number of Markov transition occurrences in digitisation steps, this probability can be upper bounded by k_b independent Poisson processes, all parametrised with the maximum exit rate exhibited in the IMC. In each Poisson process the probability of at most n Markov transition(s) firing in one digitisation step is $e^{-\lambda\delta} \sum_{i=0}^{n} \frac{(\lambda\delta)^i}{i!}$, therefore the probability of a violation of this assumption in at least one digitisation step is $1 - e^{-\lambda b}\big(\sum_{i=0}^{n} \frac{(\lambda\delta)^i}{i!}\big)^{k_b}$.

It is worthwhile to note that open and closed intervals of type $(0, b]$ and $[0, b]$ are treated in the same manner based on Theorem 2. They lead to the same fixed point computation of time bounded reachability, in contrast to bounded until [30]. We can directly extend Definition 5 to intervals with non-zero lower bounds and adapt Theorem 2 accordingly.

Theorem 3. *Given IMC $\mathcal{M} = (S, Act, \longrightarrow, \dashrightarrow, s_0)$, a set of goal states $G \subseteq S$, an interval $I \in \mathcal{Q}$ with $\inf I = a > 0$, $\sup I = b > a$ and $\lambda = \max_{s \in S_M} E(s)$. Assume digitisation step $\delta > 0$ is selected such that $a = k_a \delta$ and $b = k_b \delta$ for some $k_a, k_b \in \mathbb{N}$ and $n > 0$ is the order of approximation. For all $s \in S$ it holds*

$$p^{\mathcal{M}_\delta(n)}_{\max}(s, \Diamond^I G) - \Big(1 - e^{-\lambda a}\Big(\sum_{i=0}^{n} \frac{(\lambda\delta)^i}{i!}\Big)^{k_a}\Big) \leq p^{\mathcal{M}}_{\max}(s, \Diamond^I G) \leq p^{\mathcal{M}_\delta(n)}_{\max}(s, \Diamond^I G) + \Big(1 - e^{-\lambda b}\Big(\sum_{i=0}^{n} \frac{(\lambda\delta)^i}{i!}\Big)^{k_b}\Big)$$

The proof of Theorem 3 combines the one of Theorem 2 and of [26, Theorem 6.4]. It is worth noting that the digitisation error decreases by decreasing digitisation step δ or increasing the order of approximation n. Further, the error vanishes as n goes to infinity or δ goes to zero.

Improved algorithm. In this section we describe how the result of Theorem 2 and 3 can improve the original time bounded reachability approximation [29]. The structure of the algorithm remains unchanged, but is parametrised with natural n. It computes $p_{\max}^{\mathcal{M}_\delta(n)}$ as the approximation of the maximum reachability probability.

Our objective is to compute maximum probability to reach a set of goal states within a given step interval. First we restrict ourselves to the case that the lower bound of the step interval is zero. Afterwards, we extend it to the general case. Let $\mathcal{M}$ be an IMC, $G \subseteq S$ be a set of goal state and $I \in \mathcal{Q}$ be a nonempty interval with $\inf I = 0$ and $\sup I = b$. Assume digitisation step $\delta > 0$ is selected such that $b = k_b\delta$ for some $k_b \in \mathbb{N}$. We use $p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^I G)$ to denote the approximate maximum probability of reaching the goal states inside I where we only consider up to n Markov transition firing(s) within each digitisation step. Let $\mathit{Reach}^i(s)$ be the set of states that can be reached from s by only using interactive transitions.

The overall algorithm is depicted in Algorithm 1. It proceeds by backwards unfolding the IMC in an iterative manner, starting from the goal states. At the beginning, all goal states are made absorbing: all of their transitions are removed, and replaced by a digitised Markov self loop (a transition to a Dirac distribution over the source state). The initial value of probability vector is set to one for goal states and to zero otherwise. The algorithm then proceeds by intertwining *m-phases* and *i^*-phases* consecutively for k_b steps. In each iteration, i^*-phase and m-phase update reachability probabilities from interactive and Markov states to the set of goal states respectively. After completing i^*-phase and m-phase at the end of an iteration, the elements of $p_{\max}^{\mathcal{M}_\delta(n)}(\cdot, \diamondsuit^{I \ominus j\delta} G)$ are updated for both interactive and Markov states.

Input : $\mathcal{M}$ is the given IMC, $G \subseteq S$ is the set of goal state, I is the interval with $\inf I = 0$ and $\sup I = b$, $\delta > 0$ such that $b = k_b\delta$ for some $k_b \in \mathbb{N}$
Output: Maximum reachability probabilities starting from all states

begin
  make all $s \in G$ in $\mathcal{M}$ absorbing ;
  foreach $s \in G$ **do** $p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^{[0,0]} G) := 1$;
  foreach $s \in S \setminus G$ **do** $p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^{[0,0]} G) := 0$;
  foreach $s \in S_I$ **do**
  $p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^{[0,0]} G) := \max_{s' \in \mathit{Reach}^i(s) \cap S_M} p_{\max}^{\mathcal{M}_\delta(n)}(s', \diamondsuit^{[0,0]} G)$;
  for $j := k_b - 1$ ***to*** 0 **do**
    // m-phase ;
    foreach $s \in S_M$ **do** calculate $p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^{I \ominus j\delta} G)$ as in Definition 5;
    // i^*-phase ;
    foreach $s \in S_I$ **do**
    $p_{\max}^{\mathcal{M}_\delta(n)}(s, \diamondsuit^{I \ominus j\delta} G) := \max_{s' \in \mathit{Reach}^i(s) \cap S_M} p_{\max}^{\mathcal{M}_\delta(n)}(s', \diamondsuit^{I \ominus j\delta} G)$;
  end
end

Algorithm 1: Computing maximum step bounded reachability

Phases of an iteration. In the following we explain the functioning of i^*-phase and m-phase in more details. An i^*-phase maximises the reachability probabilities starting from interactive states to the set of goal states. By the law of total probability, this can split into two parts: (1) the probability of reaching Markov states from interactive states in zero time and (2) the probability of reaching goal states from Markov states. The latter has been computed by the m-phase directly preceding the i^*-phase under consideration. The former can be computed by a backward search in the interactive reachability graph underlying the IMC [29]. The number of transitions taken does not matter in this case, because they take zero time each. This step thus needs the set of all Markov states that are reachable from each interactive state s via an arbitrary number of interactive transitions. That set, $\mathit{Reach}^i(s) \cap S_M$, can be precomputed prior to the algorithm. From these sets, the i^*-phase selects states with maximum reachability probability. In an *m-phase*, we update the reachability probabilities starting from Markov states by taking at most n Markov transitions. This step is performed by solving the integral equation in Definition 5 for case $s \in S_M \setminus G$. Restricting the number n of Markov transitions in a digitisation step makes the integral equation in Definition 5 tractable, in contrast to Lemma 1. For instance, in the first-order approximation ($n = 1$) it is enough to consider zero or one Markov transition starting from a Markov state. Owing to this assumption the resulting model ($\mathcal{M}_\delta(1)$) is equivalent to the induced IPC ($\mathcal{M}_\delta$) from the original model with respect to digitisation step δ. For the second-order approximation we need to consider up to two Markov transitions starting from a Markov state.

Example 2. We now discuss by example how i^*- and m-phases are performed for $n = 2$. Assume Figure 2 is a fragment of an IMC $\mathcal{C}$ with a set of goal states G. Given time interval $I = [0, b]$ with $b > 0$ and digitisation step δ, the vector $p_{\max}^{\mathcal{C}_\delta(2)}(\cdot, \Diamond^{I \ominus \delta} G)$ has been computed for all states of $\mathcal{C}$. The aim is to compute $p_{\max}^{\mathcal{C}_\delta(2)}(s_0, \Diamond^{I} G)$. From Definition 5 we have:

$$p_{\max}^{\mathcal{C}_\delta(2)}(s_0, \Diamond^{I} G) = A^2_{I,2}(s_0, \mathit{\Delta}) = \int_0^\delta 2e^{-2t} A^1_{I,2}(s_1, \delta - t)\mathrm{d}t + e^{-2\delta} p_{\max}^{\mathcal{C}_\delta(2)}(s_0, \Diamond^{I \ominus \delta} G)$$

For s_1 we have $A^1_{I,2}(s_1, \delta - t) = \max\{A^1_{I,2}(s_3, \delta - t), A^1_{I,2}(s_5, \delta - t)\}$, since $\mathit{Reach}^i(s_1) \cap S_M = \{s_3, s_5\}$. From Definition 5 for s_3 and s_5 we have:

$$\begin{aligned} A^1_{I,2}(s_3, \delta - t) &= \int_0^{\delta - t} 3e^{-3t'} A^0_{I,2}(s_4, \delta - t - t')\, \mathrm{d}t' + e^{-3(\delta - t)} p_{\max}^{\mathcal{C}_\delta(2)}(s_3, \Diamond^{I \ominus \delta} G) \\ &= (1 - e^{-3(\delta - t)}) p_{\max}^{\mathcal{C}_\delta(2)}(s_4, \Diamond^{I \ominus \delta} G) + e^{-3(\delta - t)} p_{\max}^{\mathcal{C}_\delta(2)}(s_3, \Diamond^{I \ominus \delta} G) \end{aligned}$$

Similar calculations give:
$A^1_{I,2}(s_5, \delta - t) = (1 - e^{-5(\delta - t)}) p_{\max}^{\mathcal{C}_\delta(2)}(s_6, \Diamond^{I \ominus \delta} G) + e^{-5(\delta - t)} p_{\max}^{\mathcal{C}_\delta(2)}(s_5, \Diamond^{I \ominus \delta} G)$.

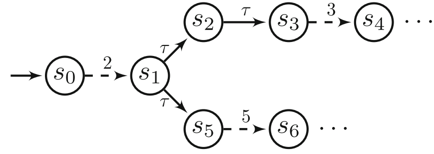


Fig. 2. An exemplary IMC fragment

Generalisation to intervals with non-zero lower bound. We can generalise time bounded reachability computation just discussed to intervals with non-zero error bound, following a recipe discussed in [2]. Assume we choose interval I such that $\inf I = a > 0$ and $\sup I = b > a$. We break the interval into two parts, first from b down to a and second from a down to zero. Within the first, we are interested in reaching one of the goal states, as a result we make the goal states absorbing. Nevertheless, within the second, it does not matter that the model is in one of the goal states, which consequently leads us to ignore goal states and reintroduce them as before. Accordingly the algorithm proceeds as follows. In the first part ($[0, b-a]$), goal states are made absorbing and reachability probabilities are computed by running Algorithm 1. The result will be used as the initial vector of the next step. Then, goal states are treated as normal states, so we undo absorbing of goal states and set $G = \emptyset$. However other calculations remain the same as before.

Complexity and efficiency. The key innovation of this approach lies in both the precision and the efficiency of the computation. Following Theorems 2 and 3, the number of iterations required to guarantee accuracy level ϵ can be calculated by determining the least k_b such that $1 - e^{-\lambda b}\big(\sum_{i=0}^{n} \frac{(\lambda\delta)^i}{i!}\big)^{k_b} \leq \epsilon$. The inequality however does not have closed-form solution with respect to k_b. Routine calculus allows us to derive that $1 - e^{-\lambda b}\big(\sum_{i=0}^{n} \frac{(\lambda\delta)^i}{i!}\big)^{k_b} \leq k_b \frac{(\lambda\delta)^{n+1}}{(n+1)!}$ which is tight in our setting, since $\lambda\delta$ is very small. Thus, we instead consider inequality $k_b \frac{(\lambda\delta)^{n+1}}{(n+1)!} \leq \epsilon$ which leads to $k_b \geq \lambda b\big(\frac{\lambda b}{(n+1)!\epsilon}\big)^{\frac{1}{n}}$. This shows how the number of iterations required to achieve a predefined accuracy level decreases by increasing the order of approximation n. In other words, using higher-order approximations gives the same error bound in less iterations.

To shed some light on this, we compare the complexity of the original first-order and the second-order instance of the novel approximation. Given accuracy level ϵ and IMC $\mathcal{M}$ as before, assume $N = |S|$ and $M = |\longrightarrow| + |\dashrightarrow|$. The best known complexity for the precomputation of set $\mathit{Reach}^i(\cdot)$ for all interactive states and hence of $\mathit{Reach}^i(\cdot) \cap S_M$ is $\mathcal{O}(N^{2.376})$ [11]. Instantiating the inequality above for $n = 2$ gives $\mathcal{O}\left(\sqrt{\frac{(b\lambda)^3}{\epsilon}}\right)$ as the complexity of the iteration count. Since the size of $\mathit{Reach}^i(s) \cap S_M$ for a given state s is at most N, the complexity of the i^*-phase is $\mathcal{O}(N^2)$. m-phase contains one step reachability computations from Markov states by considering zero, one or two Markov transitions which has the respective complexities $\mathcal{O}(N)$, $\mathcal{O}(MN)$ and $\mathcal{O}(M^2)$. Thus the resulting complexity is $\mathcal{O}\left(N^{2.376} + \left(M^2 + MN + N^2\right)\sqrt{\frac{(b\lambda)^3}{\epsilon}}\right)$, while the complexity of

the first-order approximation is $\mathcal{O}\left(N^{2.376} + (M + N^2)\frac{(b\lambda)^2}{\epsilon}\right)$ [29]. We observe that the per iteration complexity of the second-order approximation is higher, but since in almost all cases M is at least N this is a negligible disadvantage. At the same time, the number of iterations (the respective last terms) is much less. Therefore the efficiency of the second-order approximation compares favourably to the original first-order approximation, at least in theory. In the next section we compare the complexity of both algorithms in practice.

5 A Simplified Empirical Evaluation

This section reports on empirical results with an implementation that harvests the theoretical advances established, but is simplified in one dimension: Our current implementation keeps the scheduler decisions constant over each time interval of length δ, even though a timed scheduler may perform slightly better by adjusting the decision during the interval, and not at interval boundaries only. We do not yet have an error bound for the deviation introduced by this simplification. In light of the above discussion, we consider $n = 2$, thus we use a second-order approximation, and compare with the original first-order approximation.

Case study. As a case study we consider a replicated file system as it is used as part of the Google search engine [10]. The IMC specification is available as an example of IMCA tool [18]. The Google File System (GFS) splits files into chunks of equal size maintained by several chunk servers. If a user wants to access a chunk, it asks a master server which stores the address of all chunks. Then the user can directly access the appropriate chunk server to read/write the chunk. The model contains three parameters, N_{cs} is the number of chunk server, C_s is the number of chunks a server may store, and C_t is the total number of chunks.

Evaluation. We set $C_s = 5000$ and $C_t = 100000$ and change the number of chunk servers N_{cs}. The set of goal states G is defined as states in which the master server is up and there is at least one copy of each chunk available. We compute minimum and maximum time bounded reachability with respect to the set of goal states G using both the first- and the second-order approximations on different intervals of time. The former has been implemented in the IMCA tool [18], and our implementation is derived from that. All experiments were conducted on a single core of a 2.5 GHz Intel Core i5 processor with 4GB RAM running on Linux. The computation times of both algorithm under different parameter settings are reported in Table 1.

As stated before, the second-order algorithm takes less iterations for computing reachability to guarantee accuracy ϵ. The computation times reported apparently show a beneficial effect, with the speedup depending on different parameters. Table 1 indicates that the speedup gets higher with increasing λ and with increasing interval upper bounds.

Table 1. Reachability computation time in the Google file system

N_{cs}	$\|S\|$	$\|G\|$	ϵ	I	$\mathcal{M}_\delta$ time(s)		$\mathcal{M}_\delta(2)$ time(s)	
					min	max	min	max
10	1796	408	10^{-3}	$[0, 0.1]$	124.8	115.0	18.6	21.4
			10^{-3}	$[0, 0.4]$	2021.0	1823.6	145.0	165.1
			10^{-4}	$[0, 0.1]$	1308.9	1188.1	56.7	66.0
			10^{-4}	$[0.01, 0.04]$	232.8	214.0	17.1	21.9
20	7176	1713	10^{-4}	$[0, 0.01]$	319.9	308.5	52.2	54.0
			10^{-5}	$[0.005, 0.015]$	5564.9	6413.0	179.4	219.1

6 Conclusions

This paper has presented an improvement of time bounded reachability computations in IMC, based on previous work [29], which has established a digitisation approach for IMC, together with a stable error bound. We have extended this theoretical result by assuming at most n Markov transitions to fire in each digitisation step, where previously $n = 1$ was assumed. In practice, setting $n = 2$ already provides a much tighter error bound, and thus saves considerable computation time. We have demonstrated the effectiveness of the approach in our empirical evaluation with speedups of more than one order of magnitude, albeit for a simplified scheduler scenario.

Lately, model checking of *open* IMC has been studied, where the IMC is considered to be placed in an unknown environment that may delay or influence the IMC behaviour via synchronisation [9]. The approach resorts to the approximation scheme laid out in [29], which we have improved upon in the present paper. Therefore, our improvement directly carries over to the open setting. As a future work, we intend to further generalise the proposed algorithm to Markov Automata [15,14,20].

Acknowledgements. We thank Lijun Zhang for helpful discussions and comments, and Dennis Guck for developing and sharing the original implementation of the algorithm and the case study as a part of IMCA.

References

1. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Towards Performance Prediction of Verifying Continuous Time Markov Chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
2. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. IEEE Transactions on Software Engineering 29(6), 524–541 (2003)
3. Baier, C., Hermanns, H., Katoen, J.-P., Haverkort, B.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. Theoretical Computer Science 345, 2–26 (2005)

4. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
5. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
6. Boudali, H., Crouzen, P., Haverkort, B.R., Kuntz, M., Stoelinga, M.: Architectural dependability evaluation with Arcade. In: DSN, pp. 512–521. IEEE (2008)
7. Böde, E., Herbstritt, M., Hermanns, H., Johr, S., Peikenkamp, T., Pulungan, R., Rakow, J., Wimmer, R., Becker, B.: Compositional Dependability Evaluation for STATEMATE. IEEE Transactions on Software Engineering 35(2), 274–292 (2009)
8. Bravetti, M., Hermanns, H., Katoen, J.-P.: YMCA - Why Markov Chain Algebra? Electronic Notes in Theoretical Computer Science 162, 107–112 (2006)
9. Brázdil, T., Hermanns, H., Krčál, J., Křetínský, J., Řehák, V.: Verification of Open Interactive Markov Chains. In: FSTTCS, pp. 474–485 (2012)
10. Cloth, L., Haverkort, B.R.: Model Checking for Survivability. In: QEST, pp. 145–154. IEEE (2005)
11. Coppersmith, D., Winograd, S.: Matrix Multiplication via Arithmetic Progressions. In: STOC, pp. 1–6 (1987)
12. Coste, N., Hermanns, H., Lantreibecq, E., Serwe, W.: Towards Performance Prediction of Compositional Models in Industrial GALS Designs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 204–218. Springer, Heidelberg (2009)
13. Coste, N., Garavel, H., Hermanns, H., Hersemeule, R., Thonnart, Y., Zidouni, M.: Quantitative Evaluation in Embedded System Design: Validation of Multiprocessor Multithreaded Architectures. In: DATE, pp. 88–89. IEEE (2008)
14. Deng, Y., Hennessy, M.: On the Semantics of Markov Automata. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 307–318. Springer, Heidelberg (2011)
15. Eisentraut, C., Hermanns, H., Zhang, L.: On Probabilistic Automata in Continuous Time. In: LICS, pp. 342–351 (2010)
16. Esteve, M.-A., Katoen, J.-P., Nguyen, V.Y., Postma, B., Yushtein, Y.: Formal correctness, safety, dependability and performance analysis of a satellite. In: ICSE, pp. 1022–1031 (2012)
17. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
18. Guck, D.: Quantitative Analysis of Markov Automata. Master Thesis, RWTH Aachen University (2012)
19. Guck, D., Han, T., Katoen, J.-P., Neuhäußer, M.R.: Quantitative Timed Analysis of Interactive Markov Chains. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 8–23. Springer, Heidelberg (2012)
20. Hatefi, H., Hermanns, H.: Model Checking Algorithms for Markov Automata. ECEASST 53 (2012)
21. Hermanns, H.: Interactive Markov Chains. LNCS, vol. 2428. Springer, Heidelberg (2002)
22. Hermanns, H., Herzog, U., Katoen, J.-P.: Process algebra for performance evaluation. Theoretical Computer Science 274(1–2), pp. 43–87 (2002)
23. Hermanns, H., Katoen, J.-P.: Automated compositional Markov chain generation for a plain-old telephone system. Science of Computer Programming 36(1), pp. 97–127 (2000)

24. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)
25. Macià, H., Valero, V., Cuartero, F., Carmen Ruiz, M.: sPBC: A Markovian Extension of Petri Box Calculus with Immediate Multiactions. Fundamenta Informaticae 87(3–4), pp. 367–406 (2008)
26. Neuhäusser, M.R.: Model Checking Nondeterministic and Randomly Timed Systems. PhD Thesis, RWTH Aachen University and University of Twente (2010)
27. Haverkort, B.R., Kuntz, M., Remke, A., Roolvink, S., Stoelinga, M.: Evaluating repair strategies for a water-treatment facility using Arcade. In: DSN, pp. 419–424 (2010)
28. Pulungan, R.: Reduction of Acyclic Phase-Type Representations. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany (2009)
29. Zhang, L., Neuhäusser, M.R.: Model Checking Interactive Markov Chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010)
30. Zhang, L., Jansen, D.N., Nielson, F., Hermanns, H.: Automata-Based CSL Model Checking. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 271–282. Springer, Heidelberg (2011)

Appendix

Proof of Theorem 2

We present the proof of Theorem 2 in a restricted setting and afterwards briefly discuss how to extend it to cover the entirety of the theorem. We assume that $I = [0, b]$ and focus on the case $n = 2$. Lemma 1 for $s \in S_M \setminus G$ can be rewritten [26, Section 6.3.1] into

$$p^{\mathcal{M}}_{\max}(s, \diamondsuit^I G) = \int_0^\delta E(s)e^{-E(s)t} \sum_{s' \in S} \mathbf{P}(s, \bot, s') p^{\mathcal{M}}_{\max}(s', \diamondsuit^{I \ominus t} G)\, \mathrm{d}t + e^{-E(s)\delta} p^{\mathcal{M}}_{\max}(s, \diamondsuit^{I \ominus \delta} G) \tag{1}$$

The following holds from Definition 5 for $s \in S_M \setminus G$ and $n = 2$:

$$p^{\mathcal{M}_\delta(2)}_{\max}(s, \diamondsuit^I G) = A^2_{I,2}(s, \delta) = \int_0^\delta E(s)e^{-E(s)t} \sum_{s' \in S} \mathbf{P}(s, \bot, s') A^1_{I,2}(s', \delta - t)\, \mathrm{d}t + e^{-E(s)\delta} p^{\mathcal{M}_\delta(2)}_{\max}(s, \diamondsuit^{I \ominus \delta} G) \tag{2}$$

We have to prove that:

$$p^{\mathcal{M}_\delta(2)}_{\max}(s, \diamondsuit^I G) \le p^{\mathcal{M}}_{\max}(s, \diamondsuit^I G) \le p^{\mathcal{M}_\delta(2)}_{\max}(s, \diamondsuit^I G) + 1 - e^{-\lambda b}\Big(\sum_{i=0}^{2} \frac{(\lambda\delta)^i}{i!}\Big)^{k_b}$$

In the following, we prove the upper bound of the approximation. For the proof of lower bound see [26, Lemma 6.6].

Proof. The proof is by induction over k_b:
1. $k_b = 1$: We consider two cases:

a. $s \in S_M \setminus G$: Let Π^δ be the set of paths that reach G within δ time unit. In the approximation we measure the set of paths that have at most two Markovian jumps and then reach G. Let this set be denoted by $\Pi^\delta_{\leq 2}$. Since we have $\Pi^\delta = \Pi^\delta_{\leq 2} \cup \Pi^\delta_{>2}$ and $\Pi^\delta_{\leq 2}$ and $\Pi^\delta_{>2}$ are disjoint, we have: $Pr^\omega_{A,s}(\Pi^\delta) - Pr^\omega_{A,s}(\Pi^\delta_{\leq 2}) = Pr^\omega_{A,s}(\Pi^\delta_{>2})$. The probability $Pr^\omega_{A,s}(\Pi^\delta_{>2})$ can be bounded by the probability of more than two arrivals in a Poisson process with the largest exit rate appearing in the IMC within a time interval of length δ. For the Poisson process, this probability is $1 - e^{-\lambda\delta}\big(\sum_{i=0}^{2} \frac{(\lambda\delta)^i}{i!}\big)$.

b. $s \in S_I \setminus G$: This case reduces to case 1.a as follows. We have

$$
\begin{aligned}
p^{\mathcal{M}}_{\max}(s, \diamondsuit^{I\ominus t} G) &= \max_{s' \in Reach^i(s) \cap S_M} p^{\mathcal{M}}_{\max}(s', \diamondsuit^{I\ominus t} G) \\
p^{\mathcal{M}_\delta(2)}_{\max}(s, \diamondsuit^{I\ominus t} G) &= \max_{s' \in Reach^i(s) \cap S_M} p^{\mathcal{M}_\delta(2)}_{\max}(s', \diamondsuit^{I\ominus t} G)
\end{aligned}
$$

From the above equations there exists $s' \in S_M$ such that $p^{\mathcal{M}}_{\max}(s, \diamondsuit^I G) = p^{\mathcal{M}}_{\max}(s', \diamondsuit^I G)$. Because s' is a Markov state, the upper bound for s' is deployed to s.

2. $k_b - 1 \rightsquigarrow k_b$: We assume the upper bound holds for $k_b - 1$:

$$
p^{\mathcal{M}}_{\max}(s, \diamondsuit^{I\ominus\delta} G) \leq p^{\mathcal{M}_\delta(2)}_{\max}(s, \diamondsuit^{I\ominus\delta} G) + 1 - e^{-\lambda(k_b-1)\delta}\Big(\sum_{i=0}^{2} \frac{(\lambda\delta)^i}{i!}\Big)^{k_b-1} \tag{3}
$$

Assume $B^i(s,t) = p^{\mathcal{M}}_{\max}(s, \diamondsuit^{I\ominus t} G) - A^i_{I,2}(s, \delta - t)$ for $0 \leq t \leq \delta,\ i = \{0,1,2\}$ and $C(s) = p^{\mathcal{M}}_{\max}(s, \diamondsuit^{I\ominus\delta} G) - p^{\mathcal{M}_\delta(2)}_{\max}(s, \diamondsuit^{I\ominus\delta} G)$. We consider two cases:

a. $s \in S_M \setminus G$: From Eq. 1 and 2 we have:

$$
\begin{aligned}
B^2(s,0) &= p^{\mathcal{M}}_{\max}(s, \diamondsuit^I G) - p^{\mathcal{M}_\delta(2)}_{\max}(s, \diamondsuit^I G) \\
&= \int_0^\delta E(s) e^{-E(s)t} \sum_{s' \in S} \mathbf{P}(s, \perp, s') B^1(s', t)\, \mathrm{d}t + e^{-E(s)\delta} C(s)
\end{aligned} \tag{4}
$$

We try to find an upper bound for $B^1(s',t)$ for $s' \in S_M$:

$$
\begin{aligned}
B^1(s',t) &= p^{\mathcal{M}}_{\max}(s', \diamondsuit^{I\ominus t} G) - A^1_{I,2}(s', \delta - t) \\
&= \int_0^{\delta - t} E(s') e^{-E(s')\tau} \sum_{s'' \in S} \mathbf{P}(s', \perp, s'') B^0(s'', t + \tau)\, \mathrm{d}\tau \\
&\quad + e^{-E(s')(\delta - t)} C(s')
\end{aligned} \tag{5}
$$

Now we find an upper bound for $B^0(s'', t+\tau)$. For $s'' \in S_M$ we have:

$$\begin{aligned}
B^0(s'', t+\tau) &= p_{\max}^{\mathcal{M}}(s'', \Diamond^{I \ominus (t+\tau)} G) - A_{I,2}^0(s', \delta - t - \tau) \\
&= \int_0^{\delta - t - \tau} E(s'') e^{-E(s'')u} \sum_{v \in S} \mathbf{P}(s'', \bot, v) p_{\max}^{\mathcal{M}}(v, \Diamond^{I \ominus (t+\tau+u)} G)\, \mathrm{d}u \\
&\quad + e^{-E(s')(\delta - t - \tau)} p_{\max}^{\mathcal{M}}(s'', \Diamond^{I \ominus \delta} G) - p_{\max}^{\mathcal{M}_\delta(2)}(s'', \Diamond^{I \ominus \delta} G) \\
&= \int_0^{\delta - t - \tau} E(s'') e^{-E(s'')u} \sum_{v \in S} \mathbf{P}(s'', \bot, v) p_{\max}^{\mathcal{M}}(v, \Diamond^{I \ominus (t+\tau+u)} G)\, \mathrm{d}u \\
&\quad - (1 - e^{-E(s')(\delta - t - \tau)}) p_{\max}^{\mathcal{M}_\delta(2)}(s'', \Diamond^{I \ominus \delta} G) \\
&\quad + e^{-E(s'')(\delta - t - \tau)} C(s'')
\end{aligned} \tag{6}$$

We know that:

$$\int_0^{\delta - t - \tau} E(s'') e^{-E(s'')u} \sum_{v \in S} \mathbf{P}(s'', \bot, v) p_{\max}^{\mathcal{M}}(v, \Diamond^{I \ominus (t+\tau+u)} G) \mathrm{d}u \le 1 - e^{-E(s'')(\delta - t - \tau)}$$

Plugging the above inequality and 3 into 6 gives:

$$B^0(s'', t+\tau) \le 1 - e^{-\lambda(k_b \delta - t - \tau)} \Big(\sum_{i=0}^{2} \frac{(\lambda\delta)^i}{i!} \Big)^{k_b - 1} \tag{7}$$

Plugging 3 and 7 into 5 gives:

$$B^1(s', t) \le 1 - e^{-\lambda(k_b \delta - t)} \Big(\sum_{i=0}^{2} \frac{(\lambda\delta)^i}{i!} \Big)^{k_b - 1} (1 + \lambda(\delta - t)) \tag{8}$$

Note that Eq. 7 and 8 are still valid for $s', s'' \in S_I \backslash G$ with the same argument described in 1.b. Finally plugging 3 and 8 into 4 gives:

$$B^2(s, 0) = p_{\max}^{\mathcal{M}}(s, \Diamond^I G) - p_{\max}^{\mathcal{M}_\delta(2)}(s, \Diamond^I G) \le 1 - e^{-\lambda b} \Big(\sum_{i=0}^{2} \frac{(\lambda\delta)^i}{i!} \Big)^{k_b}$$

b. $s \in S_I \setminus G$: In this case the proof is similar to 1.a.

This proof can directly be extended to intervals with open bounds and to intervals with nonzero lower bounds. Furthermore it can be embedded into an induction on n, thereby showing the theorem for any natural n. We need to skip these cases because of space limitations.

Checking Compatibility of Web Services Behaviorally

Kais Klai[1] and Hanen Ochi[2]

[1] Institut TELECOM SudParis, CNRS UMR Samovar,
9 rue Charles Fourier, 91011 Evry, France
kais.klai@telecom-sudparis.eu
[2] LIPN, CNRS UMR 7030, Université Paris 13,
99 avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
hanen.ochi@lipn.univ-paris13.fr

Web services composition is an emerging paradigm for enabling application integration within and across organizational boundaries. In this context, we propose an approach based on Symbolic Observation Graphs (SOG) allowing to decide whether two (or more) web services can cooperate safely. The compatibility between two web services is defined by the well known soundness property on open workflow nets. This property guarantees the absence of anomalies (e.g. deadlock) that can appear after composition. We propose to abstract the concrete behavior of a web service using a SOG and show how composition of web services as well as the compatibility check can be achieved through the composition of their abstractions (i.e. SOGs). This approach allows to respect the privacy of the services since SOGs are based on collaborative activities only and hide the internal structure and behavior of the corresponding service.

1 Introduction

Service oriented architecture (SOA) has evolved to become a promising technology for the integration of disparate software components using Internet protocols. These components, called *Web Services*, are available in the distributed environment of the Internet. Organizations attempt to provide their own services to be matched with others following a request, their complex tasks are resolved using a combination of several web services. For automatically selecting and composing services in a well-behaved manner, information about the services has to be exposed. Usually, web services are published by giving their public description behavior in a repository, such as Universal Description, Discovery and Integration UDDI, in order to make possible the collaboration with potential requesters. In particular, this information must be sufficient to decide whether the composition of two services is possible. However, organizations usually want to hide the trade secrets of their services and thus need to find a proper abstraction which is published instead of the service itself in the repository. Thus, the public abstraction should satisfy two contradictory requirements: on one hand, it should respect the privacy of the underlying organization. On the other hand, it should supply enough information to allow the collaboration and the communication with potential partners in a correct way. Thus, correctness of the original

F. Arbab and M. Sirjani (Eds.): FSEN 2013, LNCS 8161, pp. 267–282, 2013.
DOI: 10.1007/978-3-642-40213-5_17,
© IFIP International Federation for Information Processing 2013

composite web service should be detected from the analysis of the composition of the corresponding public abstractions. Among other abstraction approaches, the Symbolic Observation Graph (SOG) based technique, initially introduced for model checking of concurrent systems [4] and then applied to the verification of inter enterprise business processes [8,11], is promising. A SOG is a graph whose construction is guided by a subset of *observed* actions. The nodes of a SOG are aggregates hiding a set of local states which are connected with non observed actions. The arcs of a SOG are exclusively labeled with observed actions. Thus, we propose to use SOGs as abstraction of web services. By observing the collaborative activities of a web service, publishing a SOG as an abstraction allows to hide its internal behavior inside the aggregates. The strength of such approach is that a SOG associated with a web service represents a reduced abstraction of its reachable state space while preserving its behavioral properties (e.g. deadlock freeness, temporal properties, ...). Checking the compatibility of two web services is reduced to check the compatibility on the composition of their SOGs.

In this paper, a web service is formally represented by an oWF-net [14]. Two web services are said to be compatible if the composite oWF-net is sound [17]. The *soundness* property on a oWF-net is defined by three requirements: (1) *option to complete*: starting from any reachable state, it is possible to reach a final state, (2) *proper completion*: there is no reachable state strictly greater than a final state, and, (3) *no dead transitions*: each action is executed at least in one reachable state. Although, in practice, the behavior of web services is frequently described using industrial description languages such as BPEL4WS, BPWL and WSCI, several approaches allow to map these models to the formal description languages [13][6] (Petri nets). Thus, our approach is relevant for a very broad class of modeling languages and we can use an UDDI registry as a repository to extract web service's specifications for this purpose.

This paper is organized as follows: first, Section 2 presents some preliminary notions on oWF-nets, their composition and the notion of *soundness*. Then, a running example is presented in Section 3 allowing to illustrate our approach through the paper. In Section 4, we present symbolic observation graphs and how the soundness property is preserved by such an abstraction. Composition of SOGs and checking the compatibility property is the issue of Section 5. In Section 6, we discuss some related works. Finally, Section 7 concludes the paper and presents some aspects of the future work.

2 Preliminaries

2.1 Description Models

Petri nets The need for formal methods and software tools for describing and analyzing web services is widely recognized. Petri nets [15], a well known formalism for modeling real-time systems, can be used for describing and analyzing the behavior of web services.

Definition 1. *A Petri net is 4-tuple $N = \langle P, T, F, W\rangle$ where:*

- *P is a finite set of places (circles) and T a finite set of transitions (squares) with $(P \cup T) \neq \emptyset$ and $P \cap T = \emptyset$,*
- *A flow relation $F \subseteq (P \times T) \cup (T \times P)$,*
- *$W : F \rightarrow \mathbb{N}^+$ is a mapping that assigns a positive weight to any arc.*

Each node $x \in P \cup T$ of the net has a pre-set and a post-set defined respectively as follows: ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$, and $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. Adjacent nodes are then denoted by ${}^\bullet x^\bullet = {}^\bullet x \cup x^\bullet$. The incidence matrix C associated with the net is defined as follows : $\forall (p, t) \in P \times T : C(p, t) = W(t, p) - W(p, t)$

A marking of a Petri net N is a function $m : P \rightarrow \mathbb{N}$. The initial marking of N is denoted by M_0. The pair (N, M_0) is called a Petri net system.

A transition t is said to be enabled by a marking m (denoted by $m \xrightarrow{t}$) iff $\forall p \in {}^\bullet t$, $W(p, t) \leq m(p)$. If a transition t is enabled by a marking m, then its firing leads to a new marking m' (denoted by $m \xrightarrow{t} m'$) s.t. $\forall p \in P : m'(p) = m(p) + C(p, t)$. Given a set of markings S, we denote by $Enable(S)$ the set of transitions enabled by elements of S. The set of markings reachable from a marking m in N is denoted by $R(N, m)$. The set of markings reachable from a marking m, by firing transitions of a subset T' only is denoted by $Sat(m, T')$. By extension, given a set of markings S and a set of transitions T', $Sat(S, T') = \bigcup_{m \in S} Sat(m, T')$. For a marking m, $m \not\rightarrow$ denotes that m is a dead marking, i.e., $Enable(\{m\}) = \emptyset$.

oWF-nets We define a web service by its behavior and its interface. An instance of a given service corresponds to an execution of this service. The interface consists of a set of ports. A pair of ports can be connected using a channel, thus enabling the exchange of messages sent or received by services. A web service can be viewed as a control structure describing its behavior according to an interface to communicate asynchronously with other services in order to reach a final state (i.e. a state representing a proper termination). We use a particular Petri net for modeling the control-flow dimension of a web service, called *open Work-Flow net (oWF-net)* and introduced in [14]. It is essentially a liberal version of workflow nets [1], enriched with communication places representing the interface. Each communication place models a channel to send (receive) messages to (from) another oWF-net. Transitions in a oWF-net correspond to activities and places represent pre-conditions for activities.

Definition 2. *An open workflow net (oWF-net for short) is defined by a tuple $N = \langle P, T, F, W, m_0, I, O, \Omega\rangle$ where:*

- *$\langle P \cup I \cup O, T, F, W\rangle$ is a Petri net;*
- *m_0 is the initial marking;*
- *I (resp. O) is a set of input (resp. output) places ($I \cup O$ represents the set of interface places) satisfying:*
  - *$(I \cup O) \cap P = \emptyset$*
  - *$\forall p \in I : {}^\bullet p = \emptyset$ (input interfaces places)*

  - $\forall p \in O : p^{\bullet} = \emptyset$ *(output interface places)*
- *Ω is a set of final markings.*

From now on, given an oWF-net N, the subnet $N^* = \langle P, T, F^*, W^* \rangle$ is called the *inner net* of N. F^* and W^* are derived (by projection) from F and W, respectively, by removing the input and the output interface places of N.

Based on the notion of oWF-nets, we have to analyze the behavior of web services from the local point of view. So we can check the soundness property [17] to detect the anomalies on web services. The *soundness* property on a oWF-net N concerns its inner Petri net N^* and is defined by three requirements: (1) *option to complete*: starting from any reachable marking, it is possible to reach a final marking, (2) *proper completion*: there is no reachable marking strictly greater than a final marking, and, (3) *no dead transitions*: each transition is firable at least in one reachable marking.

Definition 3. *Let $N = \langle P, T, F, W, m_0, I, O, \Omega \rangle$ be an oWF-net. N is sound iff the following requirements are satisfied:*

- *option to complete: $\forall m \in R(N^*, m_0)$, $\exists m_f \in \Omega$ s.t. $m_f \in R(N^*, m)$;*
- *proper completion: $\forall m \in R(N^*, m_0)$, $\forall m_f \in \Omega \; m \geq m_f \implies m = m_f$;*
- *no dead transitions: $\forall t \in T$, $\exists m \in R(N^*, m_0)$ s.t. $m \xrightarrow{t}$.*

Composition of web services The basic web services infrastructure provides simple interactions between a client and a web service. However, the implementation of a web service's business needs generally the invocation of other web services. Thus it is necessary to combine the functionalities of several web services. The process of developing a composite service is called service composition.

Composite services are recursively defined as an aggregation of elementary and composite services. The composition of two or more services generates a new service providing both the original behavior of initial services and a new collaborative behavior for carrying out a new composite task. From modeling point of view, a composite service can be described as a recursive composition of oWF-nets. Communication between services takes place by exchanging messages via interface places. Thus, composing two oWF-nets is modeled by merging their respective shared constituents which are the equally labeled input and output interface places. Such a fused interface place models a channel and a token on such a place corresponds to a pending message in the respective channel. As it is convenient to require that all communications are bilateral and directed, i.e., every interface place $p \in (I \cup O)$ has only one oWF-net that sends into p and only one oWF-net that receives from p. Thereby, oWF-nets involved in a composition are pairwise *interface compatible.*

Definition 4. *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two oWF-nets with pairwise disjoint constituents except for interfaces. If only input places of one oWF-net overlap with output places of the other oWF-net, i.e., $I_1 \cap I_2 = \emptyset$ and $O_1 \cap O_2 = \emptyset$, then $\mathcal{N}_1$ and $\mathcal{N}_2$ are* interface compatible.

Definition 5. *Let $\mathcal{N}_i = \langle P_i, T_i, F_i, W_i, m_{0i}, I_i, O_i, \Omega_i \rangle$, for $i \in \{1, 2\}$, be two interface compatible oWF-nets. Their composition, namely $\mathcal{N}_1 \oplus \mathcal{N}_2$, is the oWF-net $N = \langle P, T, F, W, m_0, I, O, \Omega \rangle$ defined as follows:*

- $P = P_1 \cup P_2$, $T = T_1 \cup T_2$, $F = F_1 \cup F_2$, $W = W_1 \oplus W_2$
- $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$,
- $m_0 = m_{01} \oplus m_{02}$ *and* $\Omega = \Omega_1 \oplus \Omega_2$.

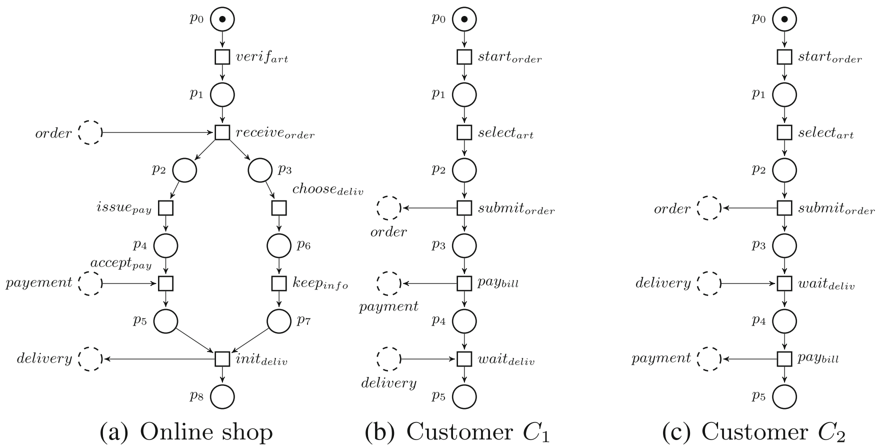
The oWF-net composition is commutative and associative i.e. for interface compatible oWF-nets $\mathcal{N}_1$, $\mathcal{N}_2$ and N_3: $\mathcal{N}_1 \oplus \mathcal{N}_2 = \mathcal{N}_2 \oplus \mathcal{N}_1$ and $(\mathcal{N}_1 \oplus \mathcal{N}_2) \oplus N_3 = \mathcal{N}_1 \oplus (\mathcal{N}_2 \oplus N_3)$. An oWF-net with an empty interface ($I = \emptyset$ and $O = \emptyset$) is called a *closed net*.

A composite web service modeled as a closed net is a service that consists of the coordination of several conceptually autonomous but interface compatible services (open nets). Although, it is not easy to specify how this coordination should behave, we focus here on semantic compatibility between web services.

Definition 6. *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two interface compatible open nets and let $N = \mathcal{N}_1 \oplus \mathcal{N}_2$. Then, $\mathcal{N}_1$ is said to be **compatible** with $\mathcal{N}_2$ iff N is sound.*

3 Running Example

Throughout this paper, we use an example of three web services, inspired from [16] (see Figure 1): an online shop and two different customers. The example is modeled using oWF-nets. The dashed circles denote the interface places (input/output places). While browsing an online shop, the first customer (Figure 1(b)) selects items he is interested in, pays his bill and proceeds for delivery step. For the online shop (Figure 1(a)), once the order is submitted,



(a) Online shop (b) Customer C_1 (c) Customer C_2

Fig. 1. The oWF-nets of an online shop and two customers

the subsequent payment handling and the verification process of delivery are triggered. These two tasks can be done concurrently. After verifying information about payment, the order is automatically delivered. Figure 1(c) represents a customer who behaves in a different way, since he pays his bill only after receiving the goods already bought. Note that all these oWF-nets are sound locally, and that the online shop's model is interface compatible with both customers' models.

4 Symbolic Observation Graph

4.1 Abstraction of web services

In this section we propose to use symbolic observation graphs (SOG) [4,12] in order to abstract oWF-nets. Exposing a SOG related to a service allows to hide internal activities while checking compatibility is still possible using locally computed information. Before we give the definition of a SOG, let us give some basic notations.

Observed actions Given an oWF-net N, we distinguish the transitions connected to the interface places, called *interface transitions*, from the internal transitions. The first are called *observed transitions* while the last are called *unobserved transitions*.

Definition 7. *Let $N = \langle P, T, F, W, m_0, I, O, \Omega \rangle$ be an oWF-net. The sets of observed transitions (Obs) and unobserved transitions (UnObs) are respectively defined as follows:*

- $Obs = \{t \in T \mid ({}^{\bullet}t \cup t^{\bullet}) \cap (I \cup O) \neq \emptyset\}$,
- $UnObs = T \setminus Obs$.

Observed behavior Given an oWF-net N, the *observed behavior* is defined as a mapping applied on the reachable markings, $R(N^*, m_0)$, of the inner net N^*. It is then extended progressively to sets of states. It will be established that the *observed behavior* is the necessary and sufficient local information to be retained so that compatibility between two web services can be checked. For this purpose, and for the remaining part of this paper, we assume an additional *virtual* observed transition **term** belonging to Obs. Observing **term** means that the system properly terminates. In the following, we denote by $Sat(S)$ the set of markings reachable from a marking $m \in S$, by firing only unobserved transitions (i.e., $Sat(S, UnObs)$).

Definition 8. *Let $N = \langle P, T, F, W, m_0, I, O, \Omega \rangle$ be an oWF-net. The observed behavior is progressively defined by :*

1. $\lambda_{\mathcal{N}} : R(N^*, m_0) \rightarrow 2^{Obs}$

$$\lambda_{\mathcal{N}}(m) = \begin{cases} \bullet \ (Enable(Sat(m)) \cap Obs) \cup \{term\} & \text{if } Sat(m) \cap \Omega \neq \emptyset \\ \bullet \ Enable(Sat(m)) \cap Obs & \text{otherwise} \end{cases}$$

2. $\lambda_{\mathcal{N}} : 2^{(R(N^*,m_0))} \rightarrow 2^{2^{Obs}}$
   $\lambda_{\mathcal{N}}(S) = \{\lambda_{\mathcal{N}}(m) \mid m \in S\}$
3. $\lambda_{min} : 2^{R(N^*,m_0)} \rightarrow 2^{2^{Obs}}$
   $\lambda_{min}(S) = \{X \in \lambda_{\mathcal{N}}(S) \mid \nexists Y \in \lambda_{\mathcal{N}}(S) :\ Y \subset (X \setminus \{term\})\}$

Informally, for each marking m in $R(N^*, m_0)$, the observed behavior of m, $\lambda_{\mathcal{N}}(m)$, represents the set of observed actions which can be executed from m, possibly via a sequence of unobserved actions. In addition, *term* is a member of $\lambda_{\mathcal{N}}(m)$ if and only if a final marking is reachable from m using unobserved actions only. The observed behavior $\lambda_{\mathcal{N}}$ associated with a set of markings S is a set of sets of observed actions. This set contains the observed behavior of the markings of S. Finally, the observed behavior mapping λ_{min} applied to a set of markings S is the minimal set of subsets (w.r.t. the set inclusion relation) of $\lambda_{\mathcal{N}}(S)$. The inclusion relation does not concern the *term* action. For instance, if there exist two markings $m, m' \in S$ such that $\lambda_{\mathcal{N}}(m) = \emptyset$ and $\lambda_{\mathcal{N}}(m') = \{term\}$, then both sets $\emptyset$ and $\{term\}$ will belong to $\lambda_{min}(S)$. This way we distinguish a dead marking from a final marking reached in S.

From now on, a state (marking) m is said to be dead if and only if its observed behavior is the empty set. This generalizes the original definition of a dead state since a terminal livelock (a livelock from which no observed action is enabled) is considered as a deadlock as well.

Symbolic Observation Graph The construction of the *SOG* corresponding to an oWF-net is guided by the set of observed transitions. A SOG is defined as a graph where each node is a set of markings linked by unobserved transitions and each arc is labeled by an observed transition. Nodes of the *SOG* are called *aggregates* and may be represented and managed efficiently using decision diagram techniques (BDDs, see e.g., [2]). In practice, due to the small number of observed transitions in loosely coupled oWF-nets, the SOG has a very moderate size and thus the time complexity of the verification process is negligible in comparison to the building time of the SOG (see [4,10,9,8] for experimental results). Before we define the SOG, let us define what an aggregate is.

Definition 9. *Let $N = \langle P, T = Obs \cup UnObs, F, W, m_0, I, O, \Omega\rangle$ be an oWF-net. An* aggregate *of N is a couple $a = \langle S, \lambda\rangle$ defined as follows:*

1. *S is a nonempty subset of $R(N^*, m_0)$ s.t.: $m \in S \Leftrightarrow Sat(m) \subseteq S$;*
2. *$\lambda = \lambda_{min}(S)$.*

From now on, $a.S$ and $a.\lambda$ denote the attributes of a given aggregate a. Note that the observed behavior attached to an aggregate allows to determine whether it is a final aggregate or not and whether it contains a deadlock or not. Indeed, an aggregate a contains a dead marking iff $\emptyset \in a.\lambda$. It contains a final marking iff $\exists Q \in a.\lambda$: $term \in Q$. In practice, since an aggregate is represented by a BDD, the computation of the corresponding observed behavior should be performed symbolically (using sets operations). A symbolic algorithm for the computation of the observed behavior is proposed in [7].

Definition 10. *A* symbolic observation graph *(SOG($\mathcal{N}$) for short) is a 5-tuple $\langle \mathcal{A}, Act, \rightarrow, a_0, \Omega' \rangle$ associated with an oWF-net $N = \langle P, T, F, W, m_0, I, O, \Omega \rangle$, s.t. $T = Obs \cup UnObs$, where:*

1. *$\mathcal{A}$ is a finite set of aggregates satisfying:*
   - *If for some $a \in \mathcal{A}$ and $t \in Obs$ the set $Ext(a,t) = \{m' \notin a.S \mid \exists m \in a.S, m \xrightarrow{t} m'\}$ is not empty, then there exist non-empty pairwise disjoint sets $S_1 \ldots S_k$ s.t. $Ext(a,t) = S_1 \cup \ldots S_k$, and $\forall i = 1 \ldots k$, there exists an aggregate $a_i \in \mathcal{A}$ s.t. $a_i.S = Sat(S_i, UnObs)$.*
2. *$Act = Obs$;*
3. *$\rightarrow \subseteq \mathcal{A} \times Act \times \mathcal{A}$ is the transition relation satisfying:*
   - *if $a \neq a'$, $(a,t,a') \in \rightarrow$ iff $Ext(a,t) \neq \emptyset$ and $a'.S = Sat(S', UnObs)$ for some $S' \subseteq Ext(a,t)$.*
   - *$(a,t,a) \in \rightarrow$ iff $Sat(\{m' \in R(N^*, m_0) \mid \exists m \in a.S, m \xrightarrow{t} m'\}, UnObs) = a.S$*
4. *a_0 is the initial aggregate s.t. $a_0.S = Sat(m_0, UnObs)$.*
5. *Ω' is a set of final aggregates defined by $\Omega' = \{a \in \mathcal{A} \mid a.S \cap \Omega \neq \emptyset\}$.*

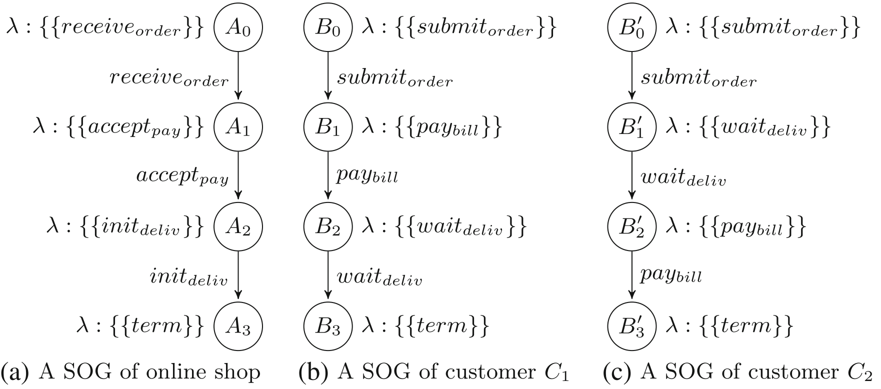
Notice that Definition 10 does not guarantee the uniqueness of a SOG for a given open net. In fact, it supplies a certain flexibility for its implementation. In particular, the SOG can be nondeterministic. It is clear that the canonical minimal SOG is obtained when the SOG is deterministic. However, one can take advantage of the nondeterminism to obtain smaller aggregates. Indeed, when two (for instance) states within an aggregate a enable an observed transition t_o, then a has one successor a' if the SOG is deterministic and two successors a'_1 and a'_2 (s.t. $a'_1 \cup a'_2 = a'$) if not. Thus, even if the SOG obtained by this way has more aggregates, its construction might consume less time and memory (aggregate's size is smaller). Our definition generalizes the one given in [12]. The construction algorithm given in [4] is an implementation where the obtained graph is deterministic.

Figure 2 shows the SOGs associated with the oWF-nets of Figure 1. Figure 2(a) shows the SOG of the online shop model while Figure 2(b) (respectively Figure 2(c)) illustrates the SOG of the customer model C_1 (respectively C_2). Each aggregate is annotated with the corresponding observed behavior and one can see that all these SOGs are sound. Moreover, the SOG of the online shop is the one which most abstract the behaviors of the original model since it has more local behaviors. Indeed, its reachability graph contains 12 reachable markings and 15 arcs against 4 aggregates and 3 arcs in the corresponding SOG.

In the following, we establish that the soundness of a oWF-net can be checked by analyzing the corresponding SOG. As for a marking m, the set of aggregates reachable from a given aggregate a is denoted by $R(a)$.

Theorem 1. *Let $\mathcal{G} = \langle \mathcal{A}, Act, \rightarrow, a_0, \Omega' \rangle$ be a SOG associated with an oWF-net N. N is sound iff the following requirements are satisfied:*

- *option to complete: $\forall a \in \mathcal{A}$, $\emptyset \notin a.\lambda \wedge \exists a_f \in \Omega' \mid a_f \in R(a)$.*
- *proper completion: $\forall a \in \mathcal{A}$, $\forall m \in a.S$, $\forall m_f \in \Omega$, $m \geq m_f \implies m = m_f$;*



(a) A SOG of online shop (b) A SOG of customer C_1 (c) A SOG of customer C_2

Fig. 2. SOGs of the running example modes

- *no dead transitions* : $\bigcup_{a \in \mathcal{A}} Enable(a.S) = T$

From the local point of view, the internal behaviors of a service are available. Thus, states inside aggregates can be analysed to check the soundness requirements but this should be done symbolically so that the efficiency of the BDD-based representation and management of the aggregates is preserved.

5 Synchronized Product of SOGs

5.1 Composition of SOGs

In this section, we tackle the main idea of this paper: we will define how we compose two (ore more) web services (each ignoring internal details about the other). Starting from two interface compatible oWF-nets $\mathcal{N}_1$ and $\mathcal{N}_2$ which are already locally sound, this section shows how to check their compatibility using their respective SOGs $\mathcal{G}_1$ and $\mathcal{G}_2$. Our objective is to reduce the verification of the compatibility between $\mathcal{N}_1$ and $\mathcal{N}_2$ (structure of $\mathcal{N}_1 \oplus \mathcal{N}_2$ is unavailable anyway) to the analysis of the composition of $\mathcal{G}_1$ and $\mathcal{G}_2$, namely $\mathcal{G}_1 \oplus \mathcal{G}_2$. To reach this goal, and in order to take into account the asynchronous composition between $\mathcal{N}_1$ and $\mathcal{N}_2$, we assume that each oWF-net exposes its input and output places (resp. transitions). Then we define a medium net N_{12} as an open net representing the interface between $\mathcal{N}_1$ and $\mathcal{N}_2$.

Definition 11. *Let $N_i = \langle P_i, T_i, F_i, W_i, m_{0i}, I_i, O_i, \Omega_i \rangle$, for $i = 1, 2$, be two interface compatible oWF-nets. The medium net related to $\mathcal{N}_1$ and $\mathcal{N}_2$, denoted by $N_{12} = \langle P_{12}, T_{12}, F_{12}, F_{12}, m_{012}, \Omega_{12} \rangle$, is the closed net defined as follows :*

- $P_{12} = (I_1 \cap O_2) \cup (O_1 \cap I_2)$
- $T_{12} = \{t \in T_i;\ {}^\bullet t^\bullet \cap ((I_i \cap O_j) \cup (O_i \cap I_j)) \neq \emptyset\}$ *for* $i, j \in \{1, 2\}$ *and* $i \neq j$
- $F_{12} = F_1|_{(P_{12} \times T_{12}) \cup (T_{12} \times P_{12})} \cup F_2|_{(P_{12} \times T_{12}) \cup (T_{12} \times P_{12})}$
- $W_{12} = W_1|_{F_{12}} \cup W_2|_{F_{12}}$

– $m_{012} = \{0\}$ *i.e. all places are empty*
– $\Omega_{12} = \{m_{012}\}$

The transitions of the medium net are the interface transitions of $\mathcal{N}_1$ and $\mathcal{N}_2$ while its places are their interface places.

It is clear that the set of reachable markings of the medium net is infinite. However, if we assume that the composed net $\mathcal{N}_1 \oplus \mathcal{N}_2$ is bounded, then the number of states that are reachable by the interface places is finite. If the bound of an interface place is n then this place can be in $n+1$ different states at most. Under such an assumption and knowing the bound of each place of the medium net, one can build a reachability graph that covers all the possible behaviors related to the interface places in $\mathcal{N}_1 \oplus \mathcal{N}_2$. The obtained graph is called *interface graph* and is defined as the following:

Definition 12. *Consider two oWF-nets $\mathcal{N}_1$ and $\mathcal{N}_2$ and their medium net N_{12}. For each place p_i (for $i = 1 \dots m$) of N_{12}, let n_i be the bound of p_i in $\mathcal{N}_1 \oplus \mathcal{N}_2$. For sake of simplicity, assume that each place p_i has a single input transition in_i and a single output transition out_i. Then, the interface graph is a a tuple $\langle \Gamma, Act, \rightarrow, m_0, \Omega \rangle$ s.t.:*

1. $\Gamma = \{\langle x_1, \dots x_m \rangle \mid 0 \leq x_1 \leq n_i \dots 0 \leq x_m \leq n_m\}$
2. $Act = \{in_i \mid i = 1, \dots, m\} \cup \{out_i \mid i = 1, \dots, m\}$
3. $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ *is a transition relation such that:*
   (a) $m \xrightarrow{in_i} m'$ *iff* $m'(b_i) = m(b_i) + 1 \wedge m'(b_i) \leq n_i$
   (b) $m \xrightarrow{out_i} m'$ *iff* $m'(b_i) = m(b_i) - 1 \wedge m'(b_i) \geq 0$
4. $m_0 = \langle 0, \dots, 0 \rangle$ *is the initial marking*
5. $\Omega = \{m_0\}$ *is the set of final markings*

The above definition constructs a reachability graph where each marking represents a possible configuration of the interface places of $\mathcal{N}_1$ and $\mathcal{N}_2$. The transition relation allows the evolution of the interface places' states in the following manner: a successor of a given marking is a marking where the number of tokens in one interface place has been increased or decreased (by one). Moreover, the initial marking (which is the final marking as well) is such that all the interface places are empty.

By observing all the transitions of the medium net, the interface graph of the medium net can be seen as a SOG. In this SOG, the aggregates are singletons (each reachable marking is an aggregate) and the observed behavior of each aggregate is also a singleton : the set of transitions appearing on the outgoing arcs of the corresponding marking. Finally, the set of final aggregates is again a singleton containing the initial aggregate.

Figure 4 illustrates the SOG associated with the medium net of Figure 3. The binary representation of each state number gives the state of the interface places (order, payment and delivery respectively). For instance, state number 5 stands for 101, i.e., only the interface place of payment is not marked. Unlike the SOGs associated with $\mathcal{N}_1$ and $\mathcal{N}_2$, the SOG of the medium net is not supposed to be built a priori. Thus, the bounds of the places of N_{12} are not supposed to

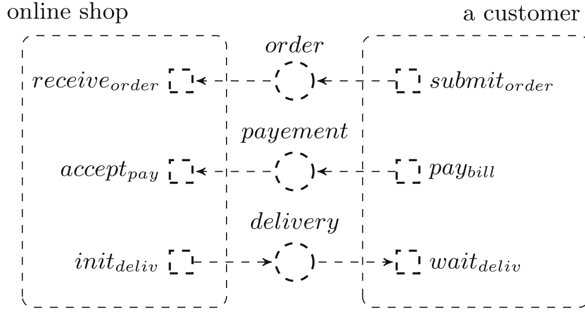


Fig. 3. The medium net of the running example

be known, as long as the composed net $\mathcal{N}_1 \oplus \mathcal{N}_2$ is bound. In the following, the SOG of the medium will be computed on-the-fly during the composition of $\mathcal{G}_1$ and $\mathcal{G}_2$. The composition of $\mathcal{G}_1$ and $\mathcal{G}_2$, denoted by $\mathcal{G}_1 \oplus \mathcal{G}_2$ is then defined as a synchronized product between three SOGs corresponding to $\mathcal{N}_1$, N_{12} and $\mathcal{N}_2$ respectively. Before we define the composition of SOGs, it is important to first show how, using observed behavior of three aggregates a_1, a_2 and a_{12} of $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_{12}$ respectively, one can compute the observed behavior of the aggregate resulting from their composition.

Note that the set of states $a.S$ of an aggregate a has not to be stored explicitly within an aggregate. Once the SOG is built, it will not play any role in the composition process. However, since our goal is to reduce the compatibility check of two oWF-nets to the analyzing of their SOGs, we need to know which are the enabled transitions (especially local transitions) in each aggregate. Given a oWF-net $N = \langle P, T, F, W, m_0, I, O, \Omega \rangle$ and an associated SOG G with respect to the set of observed transitions Obs, an aggregate of G is henceforth identified by its observed behavior λ and the set of enabled local transitions, namely E. Formally, $a.E = \{t \in T \setminus Obs \mid \exists m \in a.S,\ m\xrightarrow{t}\}$.

Definition 13. *Let $\mathcal{G}_i$, for $i = 1, 2$, be two SOGs associated with two oWF-nets and let $\mathcal{G}_{12}$ be the SOG associated with their medium net. Let a_1, a_2 and a_{12} be three aggregates of these SOGs respectively. The* product aggregate $a = (a_1, a_{12}, a_2)$ *is defined by:*

1. $a.\lambda = \{((x \cap y) \cup (x \cap (Obs_1 \setminus Obs_{12}))) \cup ((y \cap z) \cup (z \cap (Obs_2 \setminus Obs_{12}))) \mid x \in a_1.\lambda,\ y \in a_{12}.\lambda \text{ and } z \in a_2.\lambda\}$;
2. $a.E = a_1.E \cup a_2.E$

Note first that $a_{12}.\lambda$ is a singleton, that $Obs_i \cap Obs_{12}$, for $i = 1, 2$, is not empty (because $\mathcal{N}_1$ and $\mathcal{N}_2$ are interface compatible) but Obs_i is not necessarily a subset of Obs_{12}, and that $Obs_1 \cap Obs_2 = \{term\}$. When we compose a_1 and a_2, if a_1 (resp. a_2) can progress in $\mathcal{G}_1$ (resp. $\mathcal{G}_2$) by using local observed transitions (i.e., transitions in $Obs_1 \setminus Obs_{12}$ (resp. $Obs_2 \setminus Obs_{12}$)), the product aggregate a should be able to do the same. If this is not the case, then a has to have the same behavior as a_1 (resp. a_2) and a_{12} conjointly.

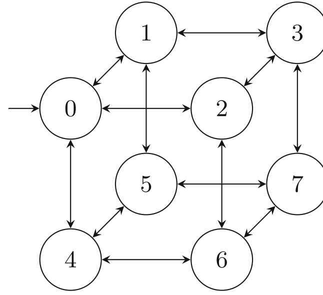


Fig. 4. Interface graph of medium net

Definition 14. *Let $\mathcal{G}_i = \langle \mathcal{A}_i, Obs_i, \rightarrow_i, a_{0i}, \Omega_i \rangle, i = 1, 2$ be two SOGs corresponding to two oWF-nets $\mathcal{N}_1$ and $\mathcal{N}_2$. Let $\mathcal{G}_{12} = \langle \mathcal{A}_{12}, Obs_{12}, \rightarrow_{12}, a_{012}, \Omega_{12} \rangle$ be the SOG of the medium net N_{12}. The composition of $\mathcal{G}_1$ and $\mathcal{G}_2$, namely $\mathcal{G}_1 \oplus \mathcal{G}_2 = \langle \mathcal{A}, Act, \rightarrow, a_0, \Omega \rangle$ is defined as follows:*

1. $\mathcal{A} \subseteq \mathcal{A}_1 \times \mathcal{A}_{12} \times \mathcal{A}_2$;
2. $Act = Obs_1 \cup Obs_2$;
3. $\rightarrow$ *is the transition relation, defined by:*
   $\forall (a_1, a_{12}, a_2) \in \mathcal{A},\ \forall (a'_1, a'_{12}, a'_2) \in \mathcal{A},\ (a_1, a_{12}, a_2) \xrightarrow{o} (a'_1, a'_{12}, a'_2) \Leftrightarrow$
   $$\begin{cases} a_1 \xrightarrow{o}_1 a'_1 \wedge a_{12} \xrightarrow{o}_{12} a'_{12} \wedge a'_2 = a_2 & \text{if } o \in (Obs_1 \cap Obs_{12}) \\ a'_1 = a_1 \wedge a_{12} \xrightarrow{o}_{12} a'_{12} \wedge a_2 \xrightarrow{o}_2 a'_2 & \text{if } o \in (Obs_2 \cap Obs_{12}) \\ a_1 \xrightarrow{o}_1 a'_1 \wedge a'_{12} = a_{12} \wedge a'_2 = a_2 & \text{if } o \in (Obs_1 \setminus Obs_{12}) \\ a'_1 = a_1 \wedge a'_{12} = a_{12} \wedge a_2 \xrightarrow{o}_2 a'_2 & \text{if } o \in (Obs_2 \setminus Obs_{12}) \end{cases}$$
4. $a_0 = (a_{01}, a_{012}, a_{02})$;
5. $\Omega = \Omega_1 \times \Omega_{12} \times \Omega_2$.

The composition of the SOGs is similar to the classical synchronized product between graphs, except the fact that nodes are aggregates (carrying additional information) instead of single states. However, the asynchronous composition of the corresponding oWF-nets has been reduced to a synchronous composition involving the medium net. The evolution in $\mathcal{G}_1 \oplus \mathcal{G}_2$ can stand for a local evolution to $\mathcal{G}_1$ (resp. $\mathcal{G}_2$) by using point 3 (resp. 4) of the transition relation in Definition 14, or a simultaneous evolution in $\mathcal{G}_1$ (resp. $\mathcal{G}_2$) and $\mathcal{G}_{12}$ by using point 1 (resp. 2). Given a local transition t in $\mathcal{N}_1$, for instance, one can check whether it remains enabled after composition or not. Indeed, the union of the E attribute of each aggregate a_1, being a part of an aggregate of $\mathcal{G}_1 \oplus \mathcal{G}_2$, should contain t. Otherwise, the transition t is not enabled by the composite net and the transition t becomes dead in the composition. If all the local transitions remain enabled, the other requirements of soundness can be deduced by analyzing the synchronized product of the SOGs.

Figure 5 illustrates the two SOGs obtained by synchronizing the SOG of online shop of Figure 2(a) with the SOGs of costumer C_1 and C_2 of Figure 2.

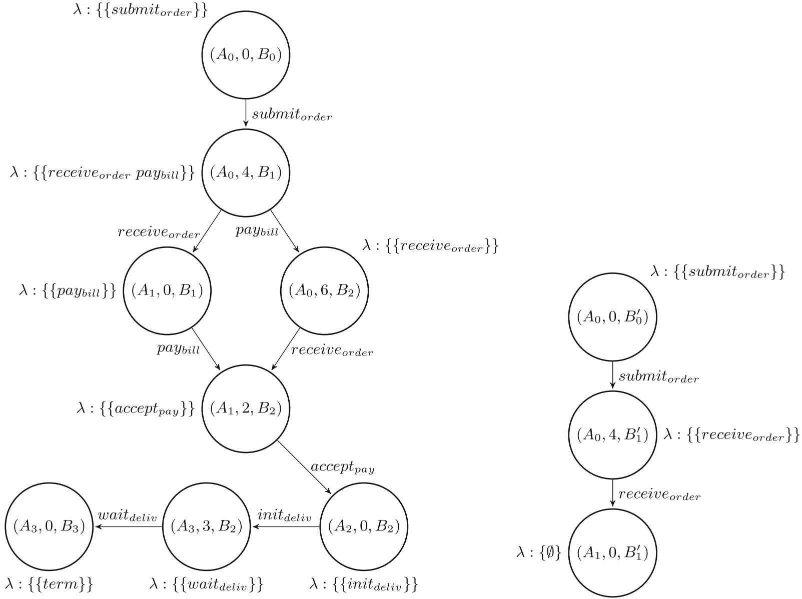


Fig. 5. the SOG synchronized product

Theorem 2. *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two oWF- nets and let $\mathcal{G}_1$ and $\mathcal{G}_2$ be the corresponding SOGs respectively. Then, $\mathcal{G}_1 \oplus \mathcal{G}_2$ is a SOG of $\mathcal{N}_1 \oplus \mathcal{N}_2$ with respect to $Obs_1 \cup Obs_2$.*

5.2 Checking Services Compatibility

Our goal is to check compatibility between two interface compatible oWF-nets $\mathcal{N}_1$ and $\mathcal{N}_2$ using their respective SOGs $\mathcal{G}_1$ and $\mathcal{G}_2$. We assume that the two oWF-nets are already sound. For checking compatibility, we have to check the soundness property of $\mathcal{N}_1 \oplus \mathcal{N}_2$. This verification will be reduced to the analysis of the synchronized product of $\mathcal{G}_1$ and $\mathcal{G}_2$, denoted $\mathcal{G}_1 \oplus \mathcal{G}_2$.

Theorem 3. *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two oWF-nets locally sound and let $\mathcal{G}_1$ and $\mathcal{G}_2$ be the corresponding SOGs respectively. Assume that all the local transitions remain enabled in the composition $\mathcal{G}_1 \oplus \mathcal{G}_2$. Then, $\mathcal{N}_1 \oplus \mathcal{N}_2$ is sound $\Leftrightarrow$*

1. *for each aggregate a in $\mathcal{G}_1 \oplus \mathcal{G}_2$, $\emptyset \notin a.\lambda$,*
2. *for each aggregate a in $\mathcal{G}_1 \oplus \mathcal{G}_2$, $\exists$ a final aggregate a_f such that $a_f \in R(a)$,*
3. *for each observed transition t, $\exists$ two aggregates a, a' in $\mathcal{G}_1 \oplus \mathcal{G}_2$ s.t. $a \xrightarrow{t} a'$.*

Corollary 1. *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two oWF-nets and let $\mathcal{G}_1$ and $\mathcal{G}_2$ be the corresponding SOGs respectively. $\mathcal{N}_1$ is compatible with $\mathcal{N}_2$ $\Leftrightarrow$ $\mathcal{G}_1 \oplus \mathcal{G}_2$ satisfies the three conditions of Theorem 3.*

By analyzing Figure 5, we can see that the composition of the online shop with the first customer is possible while it is not with the second: the corresponding composed SOG contains a deadlock (i.e., composite oWF-net not sound). For this particular example, checking the soundness property on the composition of SOGs (4 nodes and 3 edges) is easier than analyzing the original reachability graph which contain 24 nodes and 32 edges.

6 Related Work

Several approaches investigated the issue of Web services composition. Even with emergence of web service process technologies such as industrial language BPEL4WS, WSCL, etc, this specification is still not the most suitable for the verification process of compatibility behavior on composition of web services. Thus, many researchers have been interested in formal modeling and analyzing methods to better formalize the behavior of web services such as Petri net model and its variants. Authors in [5] propose a Petri net-based Algebra for modeling web services control flows. The model is expressive enough to capture the semantics of complex service combinations. Formal semantics of each composition operator (e.g. sequence, selection, refinement) is expressed by a Petri net. Using this mechanism, the analysis of web services supports the verification of web services composition by checking properties like correct termination. An other technique for modeling multiple web services interactions between BPEL processes is discussed in [19] using an extention of Petri net models called composition net (C-net). Authors analyze the model through structural properties instead of the reachability states space in order to check compatibility: the compatibility is ensured when the composite net contains a non empty minimal siphon. They impose constraints on the model to prevent it from reaching incompatible cases by using a corresponding policy based on appending additional information to channels. Then, these channels are transformed back to a BPEL description so that a new compatible web service is obtained. An other approach [3] based on mediation aided composition has been widely adopted when dealing with incompatibilities of services. In this work, given two services modeled by oWF-net, the authors propose to compose them using Mediation Transitions (MTs). They serve as information channel specifying the transferring relation of messages between different services. Then composition compatibility is verified by automatically constructing and analyzing the modular reachability graph (MRG) of the composition which is an abstraction of the original state graph. It is true that the performance of this approach is notable compared to classical ones, but MRG is represented explicitly which can be expensive.

Finally a similar approach has been introduced in [18]. In this work, the authors present a technique based on the Operating Guideline [14] for automatically checking accordance between a private view and a public view associated to each service involved in the overall process (composition of partners). A multiparty contract is specified in order to define the rules of engagement of each partner without describing its internal behavior. It can be seen as the composition of the public views from all partners. Based on the resulting contract, all

participants implement their private view on the global process in such a way that it agrees with the contract. Then, checking accordance guarantees that the process is deadlock-free and that it will always terminate properly. The main differences with our approach are: (1) this approach works only for oWF-net with acyclic behaviors (and hence deadlock freedom coincides with weak termination), (2) It is an up-down approach in the sens that it starts from a public composition (contract) whose components can be modified locally under constraints. In our case, each component ignores all about the possible partners and we also allow local changes as long as the SOG is not modified. Finally, this approach uses operating guidelines [14] to abstract services and we established in [8] that, for most cases, the SOGs-based approach is more effective in terms of memory and time consumption. In conclusion, to the best of our knowledge, none of the existing approaches combine symbolic (using BDDs) abstraction and modular verification to check the compatibility of services. They always deal with an explicit representation of the system's behavior, which accentuate the state space explosion problem.

7 Conclusion

In this paper, we proposed an approach based on a suitable model, namely Symbolic Observation Graph, to abstract web services and to analyze their composition. Such an abstraction allow to respect the privacy of each publisher by hiding service's details, and at the same time it represents the necessary information to expose on a repository for possible collaboration with other web services. We established that and how symbolic observation graphs can be extended and efficiently used for that purpose. Using such abstraction, checking compatibility between two web services (a requester and a provider) is reduced to checking compatibility on the synchronized product of the corresponding SOGs.

We are currently developing a graph-based registry for abstract web services advertisement and discovery. the next step would be to extend the presented work in order (1) to deal with other compatibility criteria (e.g., other variants of soundness, specific properties expressed with temporal logics, ...) and (2) to deal with richer models (e.g. shared resources, the explicit time).

References

1. Aalst, V.D.: The application of petri nets to workflow management (1998)
2. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
3. Du, Y., Li, X., Xiong, P.: A petri net approach to mediation-aided composition of web services. IEEE T. Automation Science and Engineering 9(2), 429–435 (2012)
4. Haddad, S., Ilié, J.-M., Klai, K.: Design and evaluation of a symbolic and abstraction-based model checker. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 196–210. Springer, Heidelberg (2004)
5. Hamadi, R., Benatallah, B.: A petri net-based model for web service composition. In: ADC 2003, pp. 191–200 (2003)

6. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to petri nets. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 220–235. Springer, Heidelberg (2005)
7. Klai, K., Desel, J.: Checking soundness of business processes compositionally using symbolic observation graphs. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE 2012. LNCS, vol. 7273, pp. 67–83. Springer, Heidelberg (2012)
8. Klai, K., Ochi, H.: Modular verification of inter-enterprise business processes. In: eKNOW, pp. 155–161 (2012)
9. Klai, K., Petrucci, L.: Modular construction of the symbolic observation graph. In: ACSD, pp. 88–97 (2008)
10. Klai, K., Poitrenaud, D.: MC-SOG: An LTL model checker based on symbolic observation graphs. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 288–306. Springer, Heidelberg (2008)
11. Klai, K., Tata, S., Desel, J.: Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 294–309. Springer, Heidelberg (2009)
12. Klai, K., Tata, S., Desel, J.: Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 294–309. Springer, Heidelberg (2009)
13. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 17–32. Springer, Heidelberg (2006)
14. Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the soa. Annals of Mathematics, Computing and Teleinformatics 1, 35–43 (2005)
15. Petri, C.A.: Concepts of net theory. In: MFCS 1973, pp. 137–146. Mathematical Institute of the Slovak Academy of Sciences (1973)
16. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding Substitutability of Services with Operating Guidelines. In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 172–191. Springer, Heidelberg (2009)
17. van der Aalst, W., van Hee, K., ter Hofstede, A., Sidorova, N., Verbeek, H., Voorhoeve, M., Wynn, M.: Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects of Computing: Applicable Formal Methods 23(3), 333–363 (2010)
18. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: Agreeing and implementing interorganizational processes. Comput. J. 53(1), 90–106 (2010)
19. Xiong, P., Fan, Y., Zhou, M.: A petri net approach to analysis and composition of web services. IEEE Transactions on Systems, Man, and Cybernetics, Part A, 376–387 (2010)

Author Index

Abdulla, Parosh Aziz 199
Ahmadi, Saeide 217
Arun-Kumar, S. 83
Atig, Mohamed Faouzi 199

Banu-Demergian, Iulia Teodora 183
Bartoletti, Massimo 66
Battisti, Luca 168
Bodeveix, Jean-Paul 111
Bonsangue, Marcello 1
Boudjadar, Abdeldjalil 111

Cimoli, Tiziana 66

De Boer, Frank 1
Delzanno, Giorgio 199
Dima, Catalin 133

Ebnenasir, Ali 17

Filali, Mamoun 111
Fokkink, Wan 217

Gaur, Manish 83
Ghassemi, Fatemeh 217

Hatefi, Hassan 250
Hermanns, Holger 250

Khosravi, Ramtin 51
Klai, Kais 267
Klinkhamer, Alex 17

Macedonio, Damiano 168
Merro, Massimo 168
Movaghar, Ali 217

Nakata, Keiko 95

Ochi, Hanen 267

Paduraru, Ciprian Ionut 183
Pinna, G. Michele 66
Podelski, Andreas 199
Pun, Ka I. 34

Ramezani, Elham 150
Rot, Jurriaan 1

Saar, Andri 95
Sabouri, Hamideh 51
Sidorova, Natalia 150
Stahl, Christian 150
Stefanescu, Gheorghe 183
Steffen, Martin 34
Stolz, Volker 34

Vaandrager, Frits 111
Vanit-Anunchai, Somsak 233
Vekris, Dimitris 133