John Hatcliff
Elena Zucca (Eds.)

LNCS 6117

Formal Techniques for Distributed Systems

Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010
Amsterdam, The Netherlands, June 2010, Proceedings

ifip

Lecture Notes in Computer Science 6117

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 of Computer Science, Saarbruecken, Germany

John Hatcliff   Elena Zucca (Eds.)

Formal Techniques for Distributed Systems

Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010
and 30th IFIP WG 6.1 International Conference, FORTE 2010
Amsterdam, The Netherlands, June 7-9, 2010
Proceedings

Springer

Volume Editors

John Hatcliff
Kansas State University, Computing and Information Sciences
234 Nichols Hall, Manhattan, KS 66502, USA
E-mail: hatcliff@ksu.edu

Elena Zucca
DISI, University of Genova
Via Dodecaneso 35, 16146 Genova, Italy
E-mail: zucca@disi.unige.it

Library of Congress Control Number: Applied for

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-13463-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-13463-0 Springer Berlin Heidelberg New York

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

springer.com

© IFIP International Federation for Information Processing 2010
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Foreword

In 2010 the international federated conferences on Distributed Computing Techniques (DisCoTec) took place in Amsterdam, during June 7-9. It was hosted and organized by the Centrum voor Wiskunde & Informatica.

DisCoTec conferences jointly cover the complete spectrum of distributed computing subjects ranging from theoretical foundations to formal specification techniques to practical considerations. The 12th International Conference on Coordination Models and Languages (Coordination) focused on the design and implementation of models that allow compositional construction of large-scale concurrent and distributed systems, including both practical and foundational models, run-time systems, and related verification and analysis techniques. The 10th IFIP International Conference on Distributed Applications and Interoperable Systems in particular elicited contributions on architectures, models, technologies and platforms for large-scale and complex distributed applications and services that are related to the latest trends in bridging the physical/virtual worlds based on flexible and versatile service architectures and platforms. The 12th Formal Methods for Open Object-Based Distributed Systems and 30th Formal Techniques for Networked and Distributed Systems together emphasized distributed computing models and formal specification, testing and verification methods.

Each of the three days of the federated event began with a plenary speaker nominated by one of the conferences. The first day Joe Armstrong (Ericsson Telecom AB) gave a keynote speech on Erlang-style concurrency, the second day Gerard Holzmann (Jet Propulsion Laboratory, USA) discussed the question "Formal Software Verification: How Close Are We?". The third and last day Joost Roelands (Director of Development Netlog) presented the problem area of distributed social data. In addition, there was a joint technical session consisting of one paper from each of the conferences and an industrial session with presentations by A. Stam (Almende B.V., Information Communication Technologies) and M. Verhoef (CHESS, Computer Hardware & System Software) followed by a panel discussion.

There were four satellite events: the Third DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS), the First International Workshop on Interactions Between Computer Science and Biology (CS2BIO) with keynote lectures by Luca Cardelli (Microsoft Research - Cambridge, UK) and Jérôme Feret (INRIA and École Normale Suprieure - Paris, France), the First Workshop on Decentralized Coordination of Distributed Processes (DCDP) with a keynote lecture by Tyler Close (Google), and the Third Interaction and Concurrency Experience Workshop with keynote lectures by T. Henzinger (IST, Austria) and J.-P. Katoen (RWTH Aachen University, Germany).

I hope this rich program offered every participant interesting and stimulating events. It was only possible thanks to the dedicated work of the Publicity Chair Gianluigi Zavattaro (University of Bologna, Italy), the Workshop Chair Marcello Bonsangue (University of Leiden, The Netherlands) and the members of the Organizing Committee, Susanne van Dam, Immo Grabe, Stephanie Kemper and Alexandra Silva. To conclude I want to thank the sponsorship of the International Federation for Information processing (IFIP), the Centrum voor Wiskunde & Informatica and the Netherlands Organization for Scientific Research (NWO).

June 2010 Frank S. de Boer

Preface

This volume contains the proceedings of the IFIP International Conference on Formal Techniques for Distributed Systems. The conference was organized as the joint activity of two conferences: FMOODS (Formal Methods for Open Object-Based Distributed Systems) and FORTE (Formal Techniques for Networked and Distributed Systems). FMOODS/FORTE was part of the federated conference event DisCoTec (Distributed Computing Techniques) 2010, which also included the 12th International Conference on Coordination Models and Languages (COORDINATION) and the 10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS).

The goal of FMOODS/FORTE is to provide a forum for fundamental research on theory and applications of distributed systems. The conference emphasizes the use of a variety of techniques for development of concurrent and distributed systems including model-based design, component and object technology, type systems, formal specification and verification, and formal approaches to testing.

The conference encourages contributions that combine theory and practice in application areas of telecommunication services, Internet, embedded and real-time systems, networking and communication security and reliability, sensor networks, service-oriented architecture, and Web services.

The FMOODS/FORTE 2010 program consisted of 13 regular and 6 short papers. These papers were selected by a 30-member Program Committee (PC) from among 38 submissions. Each paper was assigned to at least four PC members for a detailed review. Additional expert reviews were solicited if the reviews of a paper had diversified assessments or the reviewers indicated low confidence. The final decision of acceptance was based on a 10-day online discussion of the PC. The selected papers constituted a strong program of stimulating and timely topics in the areas of formal verification, algorithms and implementations, modeling and testing, process algebra and calculus, and analysis of distributed systems.

FORTE traces its heritage back to the Protocol Specification, Testing and Verification (PSTV) conference first held in 1981 in Teddington, UK. Since that first meeting, PSTV has evolved into FORTE and now into FMOODS/FORTE and this year's conference in Amsterdam was the 30th meeting in the conference series. To highlight this milestone, this year's DisCoTec plans to include a special celebration reflecting on the progress of the community since 1981 and on challenges for the future.

In the first PSTV meeting in 1981, Gerard Holzmann published a paper containing his early views of what would become the SPIN model checker. We were pleased to note this special work in the history of FORTE by featuring Dr. Holzmann as the keynote speaker of FMOODS/FORTE 2010 as part of this special celebration of PSTV - FORTE - FMOODS history.

We are deeply indebted to the PC members and external reviewers for their hard and conscientious work in preparing 154 reviews. We thank Frank de Boer, the DisCoTec General Chair, for his support, and the Steering Committees of FMOODS and FORTE for their guidance. Our gratitude goes to the authors for their support of the conference by submitting their high-quality research works. We thank the providers of the conference tool EasyChair that was a great help in organizing the submission and reviewing process.

June 2010

John Hatcliff
Elena Zucca

Organization

Program Chairs

John Hatcliff	Kansas State University, USA
Elena Zucca	University of Genoa, Italy

Program Committee

Gregor v. Bochmann	University of Ottawa, Canada
Paulo Borba	Federal University of Pernambuco, Brazil
Mario Bravetti	University of Bologna, Italy
Ana Cavalli	INT Evry, France
John Derrick	University of Sheffield, UK
Mariangiola Dezani	University of Turin, Italy
Juergen Dingel	Queens University, Kingston, Ontario, Canada
Dieter Hogrefe	University of Göttingen, Germany
Valerie Issarny	INRIA, France
Claude Jard	ENS Cachan - Bretagne, France
Einar Broch Johnsen	University of Oslo, Norway
Ferhat Khendek	Concordia University, Canada
David Lee	The Ohio State University, USA
Jay Ligatti	University of South Florida, USA
Luigi Logrippo	University of Quebec - Outaouais, Canada
Antonia Lopes	University of Lisbon, Portugal
Peter Mueller	ETH, Switzerland
Uwe Nestmann	Technical University of Berlin, Germany
Manuel Núñez	Complutense University of Madrid, Spain
Olaf Owe	University of Oslo, Norway
Alexandre Petrenko	CRIM Montreal, Canada
Frank Piessens	Katholieke Universiteit Leuven, Belgium
Arnd Poetzsch-Heffter	University of Kaiserslautern, Germany
Martin Steffen	University of Oslo, Norway
Ken Turner	University of Stirling, UK
Martin Wirsing	LMU Munich, Germany
Keiichi Yasumoto	Nara Institute of Science and Technology, Japan
Nobuko Yoshida	Imperial College London, UK

Additional Referees

Suzana Andova
Lucian Bentea
Lorenzo Bettini
Sergiy Boroday
Florent Bouchy
Roberto Bruni
Marco Carbone
Corina Cirstea
Ferruccio Damiani
Dominique Devriese
Mustafa Emre Dincturk
Jose Pablo Escobedo
Pietro Ferrara
David de Frutos-Escrig
Simon Gay
Nils Gesbert
Rohit Gheyi
Qiang Guo
Yating Hsu
Iksoon Hwang
Toqeer Israr
Juliano Iyoda
Bart Jacobs
Willy Jimenez
Ioannis Kassios
Christian Kissig
Ilham Kurnia
Felipe Lalanne
David Lee
Ugo de' Liguoro
Luis Llana
Savi Maharaj
Thi Mai Thuong Tran
Francisco Martins
Tiago Massoni
Alexandre Mota
Akio Nakata
Luca Padovani
Miguel Palomino
Andrew Phillips
Ernesto Posse
Cristian Prisacariu
Ismael Rodriguez
Fernando Rosa-Velardo
Jan Schäfer
Rudolf Schlatte
Sven Schneider
Jens Schönborn
Adenilso Simao
Carron Shankland
Jan Smans
Fausto Spoto
Volker Stolz
Dries Vanoverberghe
Mahesh Viswanathan
Frederic Vogels
Neil Walkinshaw
Yannick Welsch
Hirozumi Yamaguchi
Gianluigi Zavattaro

Table of Contents

Invited Talk

Formal Software Verification: How Close Are We? (Abstract) 1
Gerard J. Holzmann

Formal UML Modeling

Exploiting the Hierarchical Structure of Rule-Based Specifications for Decision Planning 2
Artur Boronat, Roberto Bruni, Alberto Lluch Lafuente, Ugo Montanari, and Generoso Paolillo

Reactive Semantics for Distributed UML Activities 17
Frank Alexander Kraemer and Peter Herrmann

Components and Architecture

Statistical Abstraction and Model-Checking of Large Heterogeneous Systems 32
Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay

Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude 47
Peter Csaba Ölveczky, Artur Boronat, and José Meseguer

Testing Probabilistic Distributed Systems 63
Robert M. Hierons and Manuel Núñez

Specification and Testing of E-Commerce Agents Described by Using UIOLTSs 78
Juan José Pardo, Manuel Núñez, and M. Carmen Ruiz

Testing Attribute-Based Transactions in SOC 87
Laura Bocchi and Emilio Tuosto

Joint DisCoTec Session

Grouping Nodes in Wireless Sensor Networks Using Coalitional Game Theory 95
Fatemeh Kazemeyni, Einar Broch Johnsen, Olaf Owe, and Ilangko Balasingham

Timed Process Algebra

Forgetting the Time in Timed Process Algebra: Timeless Behaviour in a Timestamped World 110
Anton Wijs

Theory and Implementation of a Real-Time Extension to the π-Calculus 125
Ernesto Posse and Juergen Dingel

Timed and Hybrid Automata

Fuzzy-Timed Automata 140
F. Javier Crespo, Alberto de la Encina, and Luis Llana

Model Checking of Hybrid Systems Using Shallow Synchronization 155
Lei Bu, Alessandro Cimatti, Xuandong Li, Sergio Mover, and Stefano Tonetta

Program Logics and Analysis

Heap-Dependent Expressions in Separation Logic 170
Jan Smans, Bart Jacobs, and Frank Piessens

Static Type Analysis of Pattern Matching by Abstract Interpretation ... 186
Pietro Ferrara

Reasoning about Distributed Systems

On-the-Fly Trace Generation and Textual Trace Analysis and Their Applications to the Analysis of Cryptographic Protocols 201
Yongyuth Permpoontanalarp

On Efficient Models for Model Checking Message-Passing Distributed Protocols 216
Péter Bokor, Marco Serafini, and Neeraj Suri

Logics for Contravariant Simulations 224
Ignacio Fábregas, David de Frutos Escrig, and Miguel Palomino

Author Index 233

Formal Software Verification: How Close Are We?

Gerard J. Holzmann

Laboratory for Reliable Software
Jet Propulsion Laboratory, California Institute of Technology
M/S 301-230, 4800 Oak Grove Drive, Pasadena, CA 91109
gholzmann@acm.org

Abstract. Spin and its immediate predecessors were originally designed for the verification of data communication protocols. It didn't take long, though, for us to realize that a data communications protocol is just a special case of a general distributed process system, with asynchronously executing and interacting concurrent processes. This covers both multi-threaded software systems with shared memory, and physically distributed systems, interacting via network channels.

The tool tries to provide a generic capability to prove (or as the case may be, to disprove) the correctness of interactions in complex software systems. This means a reliable and easy-to-use method to discover the types of things that are virtually impossible to detect reliably with traditional software test methods, such as race conditions and deadlocks.

As initially primarily a research tool, Spin has been remarkably successful, with well over one million downloads since it was first made available by Bell Labs in 1989. But our goal is te development of a tool that is not only grounded in foundational theory, but also usable by all developers of multi-threaded software, not requiring specialized knowledge of formal methods.

In this talk we try to answer the question how close we have come to reach these goals, and where especially we are still lacking. We will see that our understanding has changed of what a verification tool can do – and what it *should* do.

Keywords: Software verification, software analysis, concurrency, model checking, software testing, static source code analysis.

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, p. 1, 2010.
© IFIP International Federation for Information Processing 2010

Exploiting the Hierarchical Structure of Rule-Based Specifications for Decision Planning

Artur Boronat[1], Roberto Bruni[2], Alberto Lluch Lafuente[3], Ugo Montanari[2], and Generoso Paolillo[4]

[1] Department of Computer Science, University of Leicester, UK
[2] Department of Computer Science, University of Pisa, Italy
[3] IMT Institute for Advanced Studies Lucca, Italy
[4] Laboratorio CINI-ITEM Carlo Savy, Naples, Italy

Abstract. Rule-based specifications have been very successful as a declarative approach in many domains, due to the handy yet solid foundations offered by rule-based machineries like term and graph rewriting. Realistic problems, however, call for suitable techniques to guarantee scalability. For instance, many domains exhibit a hierarchical structure that can be exploited conveniently. This is particularly evident for composition associations of models. We propose an explicit representation of such structured models and a methodology that exploits it for the description and analysis of model- and rule-based systems. The approach is presented in the framework of rewriting logic and its efficient implementation in the rewrite engine Maude and is illustrated with a case study.

1 Introduction

Rule-based specifications have been very successful as a declarative approach in many domains. Prominent examples from the software engineering field are architectural reconfiguration, model transformation and software refactoring. One of the key success factors are the solid foundations offered by rule-based machineries like term and graph rewriting. Still, the complexity of realistic problems requires suitable techniques to guarantee the scalability of rule-based approaches. Indeed, the high number of entities involved in realistic problems and the inherently non-deterministic nature of rule-based specifications leads to large state spaces, which are often intractable.

Fortunately, many domains exhibit an inherently hierarchical structure that can be exploited conveniently. We mention among others nested components in software architectures, nested sessions and transactions in business processes, nested membranes in computational biology, and so on. In this paper we focus on the structure of *model-based* specifications due to various motivations. First, it is widely accepted that *models* enhance software comprehension [19]. Second, many model-driven development and analysis activities demand efficient and scalable approaches. Our approach aims at enhancing software comprehension by making explicit some of the structure of models, and at improving rule-based analysis techniques by exploiting such structure. For instance, the Meta-Object

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 2–16, 2010.
© IFIP International Federation for Information Processing 2010

Facility (MOF) standard defines a metamodelling paradigm by providing a set of UML-like structural modelling primitives including composition associations. Such associations impose a hierarchical structure on models. However, models are usually formalised as flat configurations (e.g. graphs) and their manipulation is studied with tools and techniques based on term rewriting or graph transformation theories [7] that do not exploit the hierarchical structure. For instance, in the MOF, models are collections of objects that may refer to other objects through references, corresponding to flat graphs in the traditional sense. In addition, some of these references are typed with composition associations in a metamodel and their semantics corresponds to structural containment. In this way, models have an *implicit* nested structure since some objects may contain other objects. To the best of our knowledge, a formalism with an *explicit* notion of structural containment has not been used for specifying model-based software artefacts yet.

In this paper we propose a formal representation of models that makes explicit the hierarchical structure of containment and a methodology that exploits such information for the description and analysis of model- and rule-based systems. The main class of analysis we shall address in this paper are planning problems that arise in various engineering activities that rely on rule-based declarations, like devising architectural reconfiguration plans, executing model transformations or taking refactoring decisions. Such problems have particular characteristics that make them different from traditional approaches. First, states in traditional planning tend to be *flat*, i.e. they typically consist of sets of ground predicates. Instead, our states are *structured* models represented by terms, offering rich descriptions that we would like to exploit conveniently. Second, rules in traditional planning are typically first-order and application conditions do not include rewrites but are limited to state predicates. Instead, our rules are *conditional* term rewrite rules à la Meseguer [14], i.e. variables can be bound to subterms and conditions can be rewrite rules whose results are used in the right-hand side. Such rules are needed to exploit structural induction during model manipulations. Third, our rules are decorated with *labels* that are used to both coordinate and guide the manipulation of models, in the spirit of *Structural Operational Semantics* [16] (SOS) and its implementation in rewriting logic [20]. We believe that the success of this discipline in the field of language semantics can be exported to model-driven transformations. Fourth, we consider *multi-criteria* optimisation where several dimensions can be used to find non-dominated optimal or near-to-optimal solutions and our approach is independent on the actual choice of quantitative aspects. This is achieved by using a generic algebraic, compositional formalism for modelling quantitative criteria, namely some variant of semirings [2], and devising plan optimisation methods that are valid for any semiring. In that way we can measure and accordingly select the most convenient model manipulations when various choices are possible, e.g. architectural reconfigurations ensuring a good load-balance but involving a low number of re-bindings, or class diagram refactorings reducing the number of classes but not requiring too many method pull-ups.

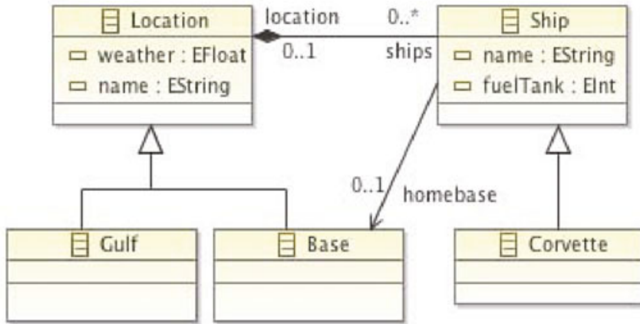


Fig. 1. Class diagram for the navy missions scenario

For this purpose we define some basic machinery based on rewriting logic and we devise a methodology to use it in practice with Maude [5], the rewrite engine tool supporting rewriting logic. In particular this paper presents 1) a novel representation of models based on nested collections of objects described with rewriting logic, 2) a methodology for exploiting the nesting structure in the declaration of rules, 3) a purely declarative presentation of planning problems with multi-criteria optimisation, i.e. we do not implement any new algorithm in Maude, but rely on Maude's reachability capabilities.

Synopsis. § 2 describes a running example, based on an industrial case study. § 3 summarises the mathematical machinery we rely on. § 4 presents the core fundamentals of our approach. § 5 explains how problem domains and instances are described and analysed. § 6 discusses related work. § 7 concludes the paper and outlines future research avenues.

2 Running Example: Navy Missions Scenario

Our running example is a naval scenario taken from a case study developed in a collaboration with the Italian company *Selex Sistemi Integrati* within the national project Tocai.It.[1] Basically, it consists of a decision support system to integrated logistic during dynamic planning of navy operations. The considered scenario consists of a naval fleet that while carrying out its current mission, is then required to switch mode of operation because some unpredictable events happened that impose new objectives with higher priority. For example, a patrol activity for peacekeeping along a coast can be required to switch to a rescue activity of civil population after a natural disaster. The re-planning requires the modelling of the new objectives and constraints that characterize the new mission and the subsequent evaluation of feasible and most convenient logistic action plan to be exploited for achieving the new goal.

[1] http://www.selex-si.com; http://www.dis.uniroma1.it/~tocai

Figure 1 depicts a simplified excerpt[2] of the class diagram for our running example, including only ingredients that we shall use throughout the paper. In particular, we see the classes for locations (Location) and ships (Ship). Two particular subclasses of locations are distinguished (Base and Gulf) as well as a particular subclass of ship (Corvette). Locations can contain ships. Ships can have a reference to their home base (homebase). Most of the classes have attributes, like a name for ships and locations (name), the fuel remaining in the tank of a ship (fuelTank) or the weather conditions for a location (weather).

3 Technical Background

Rewriting Logic. Our specifications are theories described by rewriting logic [14].

Definition 1 (rewrite theory). *A* rewrite theory $\mathcal{R}$ *is a tuple* $\langle \Sigma, E, R \rangle$ *where* Σ *is a signature, specifying the basic syntax (function symbols) and type machinery (sorts, kinds and subsorting) for terms, e.g. model descriptions;* E *is a set of (possibly conditional) equations, which induce equivalence classes of terms, and (possibly conditional) membership predicates, which refine the typing information;* R *is a set of (possibly conditional) rules, e.g. actions.*

The signature Σ and the equations E of a rewrite theory form a *membership equational theory* $\langle \Sigma, E \rangle$, whose initial algebra is $T_{\Sigma/E}$. Indeed, $T_{\Sigma/E}$ is the state space of a rewrite theory, i.e. states are equivalence classes of Σ-terms (denoted by $[t]_E$ or t for short). Usually, one is not interested in considering any term to be a state: for instance, a term can represent a part of a model like the attributes of an object. In such cases, a designated sort `State` is used and the state space of interest is then $T_{\Sigma/E,\mathtt{State}}$, i.e. all `State`-typed terms (modulo E).

Rewrite rules in rewriting logic are of the form $t \rightarrow t'$ if c, where t, t' are Σ-terms, and c is an application condition (a predicate on the terms involved in the rewrite, further rewrites whose result can be reused, memberships, etc.).

Semirings. Our specifications will be equipped with *quantitative information* such as the value of attributes or non-functional properties associated to rules. For instance, in our case study we are interested in modelling duration and risk factor of actions. There are many heterogeneous notions of quantitative features such as probabilistic, stochastic or time-related aspects, and for each one, specialised formalisms capturing their essence, e.g. Markovian models. Instead of a very specialised model, we use a generic, flexible framework for the representation of quantitative information. More precisely, we consider *semirings*, algebraic structures that have been shown to be very useful in various domains, notably in *Soft Constraint Problems* [2]. The main idea is that a semiring has a domain of partially ordered values and two operations: one for choosing the best between two values (a greatest lower bound), and another one for combining values. We focus on a particular variant of semirings, namely *constraint-semirings* (semirings, for short).

[2] The full scenario contains further entities, inheritance relations, and composition associations like fleets being made of ships, ships containing crafts, and so on.

Definition 2 (semiring). *A* semiring *is a tuple* $\langle A, \sqcup, \otimes, \mathbf{0}, \mathbf{1} \rangle$ *such that* A *is a (partially ordered) set of values;* $\mathbf{0}$ *and* $\mathbf{1}$ *are the bottom (worst) and top (best) values of* A*;* $\sqcup : A \times A \rightarrow A$ *is an operation to choose values: it is associative, commutative, and idempotent, has* $\mathbf{0}$ *as its unit element and* $\mathbf{1}$ *as its absorbing element;* $\otimes : A \times A \rightarrow A$ *is an operation to combine values: it is associative, commutative, distributes over* $\sqcup$*, has* $\mathbf{1}$ *as its unit element and* $\mathbf{0}$ *as its absorbing element. The choice operation coincides with the join operation of the lattice induced by* $a \sqsubseteq b$ *iff* $a \sqcup b = b$*.*

Notable examples are the *Boolean* ($\langle \{true, false\}, \vee, \wedge, false, true \rangle$), the *tropical* ($\langle \mathbb{R}^+, min, +, +\infty, 0 \rangle$), the *max/min* ($\langle \mathbb{R}^+, max, min, 0, +\infty \rangle$), the *probabilistic* ($\langle [0,1], max, \cdot, 0, 1 \rangle$), the *fuzzy* ($\langle [0,1], max, min, 0, 1 \rangle$), and the *set* ($\langle 2^N, \cup, \cap, \emptyset, N \rangle$) semirings. For instance, action duration is modelled in our case study with a tropical semiring. In that way, time is modelled as a positive real value, choosing between two actions means choosing the fastest one and combining two actions means adding their durations (i.e. combining them sequentially). Similarly, the risk factor is modelled with a fuzzy semiring.

Semiring based methods have a unique advantage when problems with multiple QoS criteria must be tackled: Cartesian products, exponentials and power constructions of semirings are semirings. Thus the same concepts and algorithms can be applied again and again. For instance, given two semirings C_1 and C_2 their Cartesian product $C_1 \times C_2$ is a semiring. This allows us to deal with multiple criteria at once. Moreover, such meta-operations can be implemented using Maude's parameterized modules and module operations. For example, the quantitative information regarding duration and risk of actions in our case study is modelled by the Cartesian product of the corresponding semirings.

Transition systems. The semantics of our rewrite theories are a sort of quantitative transition systems (inspired by [13]) based on the ordinary one-step semantics of rewrite theories [5].

Definition 3 (transition system). *A* quantitative transition system *is a tuple* $\langle S, \Longrightarrow, C \rangle$ *such that* S *is a set of (system) states;* C *is a semiring* $\langle A, \sqcup, \otimes, \mathbf{0}, \mathbf{1} \rangle$ *modelling the quantitative information of the system;* $\Longrightarrow \subseteq S \times A \times S$ *is a transition relation.*

We shall denote a transition (s, q, s') by $s \Longrightarrow_q s'$. We restrict our attention to finitely branching transition systems (i.e. $\forall s \in S. |\{(s, a, s') \in \Longrightarrow\}|$ is finite). The *runs of a transition system* are the (possibly infinite) paths in the underlying state transition graph, i.e. sequences $s_0 \Longrightarrow_{q_0} s_1 \Longrightarrow_{q_1} \ldots$. A finite run $s_0 \Longrightarrow_{q_0} s_1 \Longrightarrow_{q_1} \ldots s_n$ will be denoted by $s_0 \Longrightarrow^*_{\bigotimes q_i} s_n$.

Planning problems. Finally, we formalise some classic planning problems, remarking that many model-driven engineering activities like reconfiguration, refactoring or transformations can be understood as plannning problems.

Definition 4 (planning problem). *Let* $T = \langle S, \Longrightarrow, C \rangle$ *be a transition system,* $I \subseteq S$ *be a set of* initial *states and* $G \subseteq S$ *be a set of* goal *states (typically*

characterised with predicates). A planning problem *is given by the tuple* $\langle T, I, G\rangle$*. A* solution *to a planning problem* $\langle T, I, G\rangle$ *is a run* $s \Longrightarrow_q^* s'$*, such that* $s \in I$ *and* $s' \in G$*. An* optimal *or* non-dominated solution *is a solution* $s \Longrightarrow_q^* s' \in G$ *such that there is no other solution* $s_1 \Longrightarrow_{q'}^* s'_1 \in G$ *such that* $q \sqsubseteq q'$*.*

4 A Formalism for Structured Model- and Rule-Based Specifications

We present the formal means for describing model- and rule-based specifications based on the machinery of §3.

Models as nested objects. In our view, a model is a collection of possibly hierarchical objects, i.e. an object of the system may itself be a complex sub-system composed by various nested objects. The description of models is done with a signature of *nested objects* that extends Maude's object-based signature [5] with nesting features that allow for objects to contain object collections.

More precisely, our rewrite theories are based on a basic membership equational theory $\mathcal{M}_{\mathcal{N}} = \{\Sigma_{\mathcal{N}}, E_{\mathcal{N}}\}$ that provides the main signature and equations. Signature $\Sigma_{\mathcal{N}}$ is basically made of sorts $K_{\mathcal{N}}$ and operator symbols $O_{\mathcal{N}}$.

Definition 5 (basic sorts). *The set of basic sorts of* $K_{\mathcal{N}}$ *contains* `Conf`*, i.e. the sort of model configurations;* `Obj`*, i.e. the sort of objects;* `Att`*, i.e. the sort of attributes; a sort* $\texttt{Set}\{T\}$ *for each of the above sorts* T*, i.e. the sort of sets of* T*-terms;* `Oid` *of object identifiers; sort* `Cid` *of object classes.*

Sort `Conf` will be our designated `State` sort whenever we will be interested in analysing the space of possible system model configurations. We define now the symbols of the operators that build terms of the above defined sorts.

Definition 6 (basic operators). *The set of basic operator symbols* $O_{\mathcal{N}}$ *contains a constructor* $[\,\cdot\,] : \texttt{Set\{Obj\}} \to \texttt{Conf}$ *for configurations, given a set of objects; a constructor* $< \cdot : \cdot \mid \cdot \mid \cdot > : \texttt{Oid} \times \texttt{Cid} \times \texttt{Set\{Att\}} \times \texttt{Set\{Obj\}} \to \texttt{Obj}$ *for objects, such that* $<o{:}c|a|s>$ *is an object with identity* o*, class* c*, attribute set* a *and sub-objects* s*; a constant* $\texttt{none} : \to \texttt{Set}\{T\}$ *for each sort* $\texttt{Set}\{T\}$*, i.e. the empty set; a binary operator* $\cdot\,,\,\cdot : \texttt{Set}\{T\} \times \texttt{Set}\{T\} \to \texttt{Set}\{T\}$ *for each sort* $\texttt{Set}\{T\}$*, i.e. set union.*

Attribute and identifier constructors are problem-dependent, i.e. they are defined for the particular domain or instance being described. We just remark that they typically take the form $n{:}v$, where n is the attribute name and v is the attribute value. Usual attributes include references to object identifiers and quantitative information (see §3).

Example 1. Consider the diagram of Fig. 2, which illustrates an instance of our scenario where two corvettes are located at a gulf. More precisely, we see that the gulf is represented by the object of class `Gulf` with identifier 2, various attributes

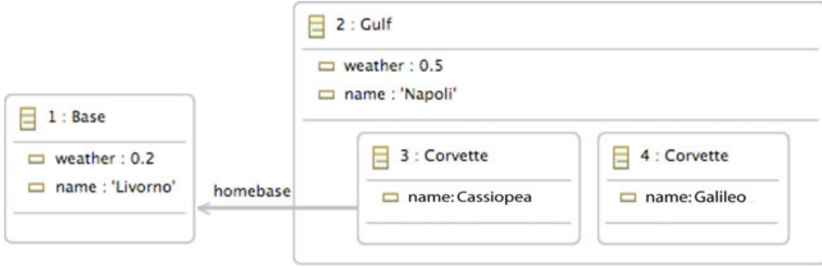


Fig. 2. A configuration with nested objects

and embedding the vessels within its area, namely objects 3 and 4, both of class Corvette. Instead, object 1 is the Base to which the corvette named Cassiopea refers as its home, by means of the reference attribute $\texttt{homebase} :\cdot : \texttt{Oid} \rightarrow \texttt{Att}$. In our notation the described scenario is denoted by term

$$\begin{array}{l}[< \texttt{1} : \texttt{Base} \mid \texttt{name} : \texttt{Livorno} \ , \ \texttt{weather} : \texttt{0.2} \mid \texttt{none} > \ , \\ \ < \texttt{2} : \texttt{Gulf} \mid \texttt{name} : \texttt{Napoli} \ , \ \texttt{weather} : \texttt{0.5} \mid \\ \qquad < \texttt{3} : \texttt{Corvette} \mid \texttt{name} : \texttt{Galileo} \mid \texttt{none} > \ , \\ \qquad < \texttt{4} : \texttt{Corvette} \mid \texttt{name} : \texttt{Cassiopea} \ , \ \texttt{homebase} : \texttt{1} \mid \texttt{none} >>]\end{array}$$

The set of equations $E_{\mathcal{N}}$ of our basic membership theory essentially axiomatises sets, i.e. it contains equations to denote the associativity, commutativity and idempotency of set union and the fact that the empty set is the identity element for set union.

Definition 7 (basic equations). *The set of basic operator symbols $E_{\mathcal{N}}$ contains equations $x \ , \texttt{none} = x$ (identity); $x \ , \ x = x$ (idempotency); $x \ , \ y = y \ , \ x$ (commutativity); $x \ , \ (y \ , \ z) = (x \ , \ y) \ , \ z$ (associativity) for each sort $\texttt{Set}\{T\}$, with $x, y, z : \texttt{Set}\{T\}$.*

Obviously, the designer might introduce new sorts, subsorting declarations or derived operators (new symbols and appropriate equations) for its own convenience, but the above presented signature is at the core of all specifications.[3]

Quantitative information. Semirings can be described with membership equational theories.

Definition 8 (semiring theory). *A* semiring theory *is a membership equational theory $\langle \Sigma, E \rangle$ such that Σ contains the sort* Cost *for carrier A, the semiring operator symbols $\sqcup, \otimes, \mathbf{0}, \mathbf{1}$, the sort of Booleans and the usual lattice operator symbols. E contains the axioms of Def. 2 and the usual equations for lattices.*

[3] Our incremental presentation does not only facilitate the reading of the paper but is supported by module importation in Maude which also has a mathematical meaning in rewriting logic, e.g. $\texttt{Set}\{T\}$ is a parametric module in Maude offering the mentioned constructors and equations.

A *concrete semiring theory* $\langle A, \sqcup, \otimes, \mathbf{0}, \mathbf{1} \rangle$ is membership equational theory that can be declared as a view of the semiring theory. For instance, The Boolean semiring theory is a view of the usual Boolean theory. This allows us to re-use Maude's predefined theories (e.g. Floating-point numbers as approximation of reals).

Conditional, labelled, quantitative rules. We are interested in rules in a particular format, namely in the style of *Structural Operational Semantics* [16] (SOS). SOS rules guarantee us a firm discipline in specifying the dynamics of a model by structural induction, i.e. by composing the transition of objects exploiting the structure of the model. In addition, we are interested in rules carrying quantitative information. More precisely, one of the rule formats the designer should follow is[4]

$$\frac{t_1 \xrightarrow{l_1}_{q_1} t_1' \qquad t_2 \xrightarrow{l_2}_{q_2} t_2'}{t_1\ ,\ t_2 \xrightarrow{l_1 \odot l_2}_{q_1 \oplus q_2} t_1'\ ,\ t_2'} \text{ if } c$$

where a transition for a configuration made of sub-parts t_1 and t_2 (of sort `Set{Obj}`) is inferred from the transitions of each of the parts. More precisely, if part i of the configuration is in state t_i and is ready to perform an l_i-labelled transition to go into state t_i' with cost q_i, then a model configuration with state $t_1\ ,\ t_2$ is ready to perform a transition labelled by $l_1 \odot l_2$ to go into state $t_1'\ ,\ t_2'$ with cost $q_1 \oplus q_2$. Eventually, some additional conditions c might be considered, but we require them to be predicates and not additional rewrites.

Operations $\odot$ and $\oplus$ are used to combine transition labels and costs, respectively. Typically, the combination of labels will follow some classical form. For instance, in synchronisation rules, l_1 and l_2 can be complementary actions, in which case $l_1 \odot l_2$ would be a silent action label τ. However, we will not make any particular choice of the synchronisation algebra. It is up to the system designer to decide which labels and label synchronisation to apply. We only remark that it is also possible to use semirings to model classical synchronisation algebras [12]. In the following we assume that our basic signature is enriched with a sort `Lab` for action labels. As for the quantitative information, the actual choice of operation $\oplus$ in each rule is up to the system designer.

In addition, a designer will be allowed to denote the transition of an object, provided that its sub-objects t are able to perform some transition:

$$\frac{t \xrightarrow{l}_{q} t'}{<o{:}C|A|t> \xrightarrow{l \odot l'}_{q \oplus q'} <o{:}C|A'|t'>} \text{ if } c$$

Such rules might affect the attributes of the container object and manipulate the action label but of course are not allowed to change the object's identifier or class. More elaborated versions of the above rule are also allowed, for instance

[4] Note that we put the rule conclusion on the bottom, the rewrite premises on top and additional conditions on a side to stick to the usual SOS notation.

involving more than one object or not requiring any rewrite of contained objects. We shall see some examples in §5.

Finally, there is a rule that is common to all specifications which allow us to derive a global step of a configuration made of a set of objects t, removing the action label, but keeping the transition cost:

$$\frac{t \xrightarrow{l}_q t'}{[t] \longrightarrow_q [t']}$$

Rewrite rules in rewriting logic are not equipped with quantitative information and that rule labels can be used only at the meta-level. This is not a problem, as there are standard techniques to encode transition annotations into states. In particular, we follow the encoding of SOS semantics in rewriting logic [20] and enrich our signature with sorts for action-prefixed states ($\texttt{Act}\{\texttt{State}\}$), a constructor $\{\cdot,\cdot\}\cdot : \texttt{Lab}\times\texttt{Cost}\times\texttt{State} \rightarrow \texttt{Act}\{\texttt{State}\}$ for action and cost prefixed states and a constructor $\{\cdot\}\cdot : \texttt{Cost} \times \texttt{State} \rightarrow \texttt{Act}\{\texttt{State}\}$ for cost prefixed states. In addition, we enforce rule application at the top-level of terms only (via Maude's `frozen` attribute) so that sub-terms are rewritten only when required in the premise of a rule (as required by the semantics of SOS rules). However, since this is basically an implementation issue, in the rest of the paper we shall continue using our notation of labelled, cost-annotated transitions, leaving implicit the fact that a rewrite $t \xrightarrow{l}_q t'$ actually denotes a rewrite $t \longrightarrow \{l, q\}t'$. In other words, quantitative information is conceptually associated to transitions, but the actual rewriting logic description constrains us to associate it to states, i.e. to use terms to represent that "a state t' was reached with cost q via an l-labelled rule".

Transition system for planning. The built-in tools of Maude allows us to explore the state space of rewrite theories. Basically, we concentrate on Maude's reachability analysis which allows us to find a rewrite sequence from a term t to a term t' satisfying some conditions. To be able to use such tools, we have to encode the analysis problems raised in §3 as rewrite theories. First, we remark that the one-step semantics of the kind of rewrite theories we are interested in is defined as follows.

Definition 9 (one-step semantics). *Let $\mathcal{R}$ be a rewrite theory equipped with a semiring C and with a designated state sort* State. *The transition system associated to $\mathcal{R}$ is $T(\mathcal{R}) = \langle S, \Longrightarrow, C\rangle$ such that $S = T_{\Sigma/E,\texttt{State}}$, i.e. states are equivalence classes of terms of sort* State*; $\Longrightarrow = \{t \Longrightarrow_q t' \mid t \longrightarrow_q t'$ is a one-step rewrite proof in $\mathcal{R}$ and t,t' are* State*-typed terms$\}$, i.e. system transitions are formed by one-step rewrites between states.*

Non-optimal Planning. Now, we concentrate on finding a solution to a planning problem $\langle T, I, G\rangle$ where T is the transition system of the rewrite theory $\mathcal{R}$ describing our specification, I is a set of initial configurations and G is the set of goal configurations. This can be done using Maude's search capabilities.

However, the presence of quantitative information introduces unnecessary redundancy. Indeed, the state space might contain duplicate states with different cost annotations. Therefore, we can forget the quantitative information just by dropping the cost annotation. Technically, this is achieved in an elegant way by introducing in $\mathcal{R}$ the equation $\{q\}t = \{\mathbf{1}\}t$. It is obvious to see that reachability in the resulting theory is enough for finding a solution of the planning problem.

Optimal Planning. Now we explain how to find optimal solutions to a planning problem via Maude's reachability analysis. The main idea is to emulate Dijkstra's shortest path algorithm, by exploring the state space of sets of non-dominated configurations. We enrich state annotations with information to explictly record the path to a state: $\{q, p\}t$ denotes that state t has been reached through path p with cost q. In addition, we use path operations like path concatenation (denoted with $\cdot$). Now, we let `Set{Conf}` be the designated sort `State` and we enrich our rewrite theory with the following rule

$$\frac{t \longrightarrow_{q'} t'}{\{q,p\}t \ , \ S \longrightarrow \{q,p\}t \ , \ \{q \otimes q', p \cdot t \Longrightarrow_{q'} t'\}t' \ , \ S} \text{ if } c$$

where c forbids the new state t' to be a dominated duplicate $(\neg\exists\{q'', p''\}t'' \in (\{q,p\}t \ , \ S) \mid t' = t'' \wedge \{q \otimes q', p \cdot t \Longrightarrow_{q'} t'\}t' \sqsubseteq \{q'', p''\}t'')$, and the state t selected for exploration to be dominated $(\neg\exists\{q'', p''\}t'' \in S \mid \{q,p\}t \sqsubseteq \{q'', p''\}t'')$. The rule basically allows us to enrich the set of discovered configurations so far in a monotonic way. Of course, we have to discard dominated configurations by introducing equation $\{q_1\}t \ , \ \{q_2\}t = \{q_1\}t$ if $q_2 \sqsubseteq q_1$.

We denote the resulting rewrite theory by $\mathcal{R}^{\mathsf{Set}}$. This provides us with a simple method for finding optimal solutions to the planning problem, as each rewrite step roughly emulates an iteration of Dijkstra's shortest-path algorithm (which can be generalised to semirings [17]).

Proposition 1 (optimal planning correctness). *Let $\mathcal{R}$ be a rewrite theory describing a system, I be the set of initial states and G be the set of goal states. Then a solution for the planning problem $\langle T(\mathcal{R}^{\mathsf{Set}}), I, G' \rangle$ is an optimal solution for the planning problem $\langle T(\mathcal{R}), I, G \rangle$, where $G' = \{S' \subseteq S \mid S' \cap G \neq \emptyset\}$.*

5 Domain and Instance Specification

This section provides some hints for describing and analysing model- and rule-based specifications in our methodology.

System domain description. System descriptions must include actual object classes and attributes. For each class C the designer must declare a sort C that represents the class and a constructor $C :\rightarrow C$. Each sort C is declared as a subsort of `Cid`. Additional subsorting relations might be added in the same spirit of class inheritance. The subsorting of classes allows us to declare rules that apply to certain classes of objects only, in a very convenient way. For instance, in

our case study we have classes for the different entities involved in the scenario, like locations (`Location`, `Gulf`, `Base`) and ships (`Ship`, `Corvette`). Sorts `Gulf` and `Base` are declared as subsorts of `Location`, and similarly for `Corvette` and `Ship`.

Next, attribute domains and constructors must be declared. Typically, attributes take the form $n{:}v$, where n is the attribute name and v is the attribute value. Typical attributes include references to object identifiers and quantitative information. In our example, for instance, we use an attribute `homebase` with domain `Oid` to allow for ships to refer to their home bases and we have an attribute `weather` with a fuzzy semiring as domain to represent the risk factor introduced by weather conditions.

Of course, the system designer might introduce additional sorts, subsorting declarations or operators (new symbols and appropriate equations) for his own convenience.

System domain rules. The domain description includes the declaration of the rewrite rules that represent the actions of the system. Some of the rules regard the actions of individual objects and are of the form:

$$<o{:}C|A|t> \xrightarrow{l}_q <o{:}C|A'|t>$$

i.e. the object o is able to perform an action with label l and cost q and the effect is reflected in its attributes. Note that such rules have no premise, i.e. they do not require any transition of sub-components. It is also usual to have unconditional rules involving more than one object (possibly with some nesting structure). For example, a basic rule of our scenario declares the ability of a ship to navigate (label `nav`) with a duration of 1 and a risk factor that depends on the weather conditions of both locations.

$$\begin{array}{l}< \mathtt{o1} : \mathtt{Location} \mid \mathtt{weather} : \mathtt{q1}\ ,\ \mathtt{a1} \mid \mathtt{t1} >\ ,\\ < \mathtt{o2} : \mathtt{Location} \mid \mathtt{weather} : \mathtt{q2}\ ,\ \mathtt{a2} \mid \mathtt{t2}\\ \qquad < \mathtt{o3} : \mathtt{Ship} \mid \mathtt{a3} \mid \mathtt{t3} >>\\ \xrightarrow{\mathtt{nav}}_{\langle 1, max\{q1,q2\}\rangle} < \mathtt{o1} : \mathtt{Location} \mid \mathtt{weather} : \mathtt{q1}\ ,\ \mathtt{a1} \mid \mathtt{t1}\ ,\\ \qquad\qquad\qquad < \mathtt{o3} : \mathtt{Ship} \mid \mathtt{a3} \mid \mathtt{t3} >>\ ,\\ \qquad\qquad < \mathtt{o2} : \mathtt{Location} \mid \mathtt{weather} : \mathtt{q2}\ ,\ \mathtt{a2} \mid \mathtt{t2} >\end{array}$$

Such individual actions are combined together with rules in the format discussed in § 4. Recall, that the actual choices of operations $\odot$ and $\oplus$ to combine action labels and quantitative information are crucial for the semantics of the rules. For instance, assume that our quantitative criteria includes action durations (which is the case of our case study) modelled with a tropical semiring. Several options are possible. If we let $\odot$ be the choice operation of the semiring (i.e. *min*) we model the fact that the fastest action is considered (in which case it is meaningful to replace t'_2 with t_2 in the conclusion of the rule). If we let $\odot$ be the join operation of the lattice underlying the semiring (i.e. *max*) we model the fact that the system has to wait to the slowest component to evolve. If we let $\odot$ be the combination operation of the the semiring (i.e. addition) we model the fact that

system components evolve sequentially. Similar choices are possible for other quantitative dimensions. It is up to the designer to choose which one is more suitable in each case. As for label synchronisation, the standard approaches are possible like Hoare (all agree on the same action label) or Milner (complementary actions are synchronised) synchronisation. In our case study we tend to use Hoare synchronisation. For instance, the rule to combine the navigation of ships is

$$\frac{t_1 \xrightarrow{\mathtt{nav}}_{q_1} t_1' \qquad t_2 \xrightarrow{\mathtt{nav}}_{q_2} t_2'}{t_1\ ,\ t_2 \xrightarrow{\mathtt{nav}}_{q_1(max,max)q_2} t_1'\ ,\ t_2'}$$

i.e. we let two sets of ships navigate together at the slowest pace and considering the worst risk factor.

System problem description. With a fixed domain description, several instances are possible. An instance can be just a term of sort `Conf` (see e.g. Example 1 in §4) denoting the initial configuration of the system, but might include instance-dependent rules as well. Usually, a problem description will include a characterisation of goal configurations. For instance, in our case study, we have specified a function `isGoal` that characterises goal configurations, namely those configurations where all ships arrive to Stromboli (to tackle the emergency due to an increase of the eruptive activity of the vulcan).

Planning activities. In order to solve planning problems the system description must be imported from one of the planning theories discussed in §4. Then we can use Maude's `search` command to perform the corresponding reachability analysis. For instance, to find a solution for the rescue problem we can execute the command

```
search [1] initialConfiguration =>* reachableConfiguration:Conf
    such that isGoal(reachableConfiguration:Conf)
```

to obtain a goal state and, subsequently, the `show path` command to obtain a solution, as a sequence of system transitions, each made of the actions need to rescue the inhabitants of Stromboli, i.e. we obtain a rescue plan.

Instead, if we want to find optimal rescue plans we have to consider the theory of configuration sets and use the command

```
search [1] initialConfiguration =>* reachableConfigurations:Set{Conf}
    such that hasGoal(reachableConfigurations)
```

in which case we might obtain an absolute optimal rescue plan (one that is fastest and with less risk) or a non-dominated rescue plan (either one that is faster but involves more risk, or one less risky but slower).

6 Related Work

We offer a brief discussion with related approaches that have influenced or inspired our work. A first source of inspiration is our previous work on *Architectural*

Design Rewriting (ADR) [4] an approach that conciliates algebraic and graph-based techniques to offer a flexible model for software systems (e.g. software architectures) and their dynamics (e.g. architectural reconfiguration). Roughly, ADR models are rewrite theories interpreted over a particular class of hierarchical graphs. Another fundamental source of inspiration is the approach of [3], which provides a rewriting logic semantics to the Meta-Object Facility (MOF) and proposes the use of rewrite rules as a declarative description of model transformations. In a way, the present work conciliates both approaches. First, by enriching the formal model of [3] with explicit hierarchical features: in [3] compositions are modelled with references, so both models are somewhat homomorphic, but our explicit representation facilitates definitions (e.g. rules or predicates) by structural induction. Second, by devising a methodology inspired by our experience in modelling and analysing with ADR.

Similar approaches can be found in the field of quantitative process algebras (e.g. with applications to software architectures [1]), rewriting logic based quantitive specifications (e.g. timed [15] or probabilistic systems [11]) or quantitative model checking (see e.g. [10]). As far as we know, the focus has been on time and probabilistic/stochastic aspects in the tradition of Markovian models. It is possible to model some of such aspects with semiring as well but in a more approximated fashion. On the other hand, semirings offer various advantages: they are compositional, not limited to two aspects and enjoy algebraic properties that are inherent to many graph exploration algorithms, starting from the well known Floyd-Warshall's algorithm to solve the all-pairs shortest path problem. A particular variant of semirings has been implemented in Maude [9]. However, the variant of semiring used is slightly different from the one we need.

Our approach is also related to AI planning and in particular action planning. Due to space constraints we cannot offer a detailed overview of such a vast research field. However, we mention some prominent approaches. A relevant planning community centers around the Planning Domain Definition Language[5] (PDDL), a meta-language to be used as common problem domain and problem description language for various planning tools. The main difference with our to describe systems with inherently hierarchical aspects and does not allow to specify flexible (e.g. conditional) rules. However, many efforts have been invested towards expressiveness and performance. Interesting branches we are currently investigating are Temporal Planning [6] which copes with planning problems with durative, concurrent actions, and Hierarchical Task Planning [18] where the execution of a (hierarchical) task might require the execution of a plan of (sub) tasks.

7 Conclusion

We have presented an approach for the description and analysis of model- and rule-based specifications with hierarchical structure. Our approach provides several benefits. First, it is built over the solid foundation of algebraic approaches

[5] http://ipc.icaps-conference.org/

like rewriting logic, structural operational semantics and semirings. Second, all the mathematical machinery is presented in the unifying, tool-supported framework of rewriting logic. Third, the approach fits perfectly with MOF-based technology as the MOF structure is somewhat homomorphic with our formalism. As a matter of fact our approach can be understood as a no-harm enhancement of the algebraic approach to MOF of [3]: one should be able to pass from a composition-as-relation representation to a composition-as-containment representation in a bijective manner, to use structural induction there where convenient. Fourth, the approach imposes a design discipline based on the hierarchical structure of composition associations, which contributes to the scalability of model- and rule-based approaches. Indeed, designers can benefit from the layered view introduced by the hierarchical structure and structured rewrite rules can lead to more efficient analysis activities.

In this regard, we are currently investigating performance comparisons between flat and structured approaches to model manipulations. In particular, preliminary results comparing classical examples of model transformations are very promising. Other current efforts regard the automatisation and tool-support for system descriptions. For instance, it should be possible to automatically derive the equational part of the theory from an UML class diagram, along the lines of the technique used in [3], e.g. deriving sorts from classes, subsorting from inheritance relations, and nesting from composition relations. In such way we could benefit from high-level front-ends.

We are also investigating more elaborated analysis problems. For instance, Maude provides a strategy language that we could use to restrict the set of acceptable plans we are interested in (e.g. by forbidding certain actions), and an LTL model checker that could be used for characterising plans with Linear-time Temporal Logic along the lines of *planning as model checking* approaches [8].

In future work we plan to conduct a comprehensive experimental evaluation to evaluate the applicability and scalability of our approach, considering automatically generated tests, industrial case studies, other application domains (e.g. architectural reconfiguration, refactoring) and comparison with state-of-the-art tools.

Acknowledgements. We are grateful to the organisers of the Dagstuhl Seminar on Graph Search Engineering (`http://www.dagstuhl.de/09491`) for their kind invitation to present some preliminary ideas, to Michele Sinisi and Raffaele Vertucci for the detailed, informal description of the case study, to the EU project Sensoria and the the Italian project Tocai.It for their support, and to Priyan Chandrapala for his prototypical hierarchical editor of EMF models.

References

1. Aldini, A., Bernardo, M., Corradini, F.: A process algebraic approach to software architecture design. Springer, Heidelberg (2010)
2. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. Journal of the ACM 44(2), 201–236 (1997)

3. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
4. Bruni, R., Lluch Lafuente, A., Montanari, U., Tuosto, E.: Style based architectural reconfigurations. EATCS 94, 161–180 (2008)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Coles, A., Fox, M., Halsey, K., Long, D., Smith, A.: Managing concurrency in temporal planning using planner-scheduler interaction. Journal on Artificial Intelligence 173(1), 1–44 (2009)
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (March 2006)
8. Giunchiglia, F., Traverso, P.: Planning as model checking. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS, vol. 1809, pp. 1–20. Springer, Heidelberg (2000)
9. Hölzl, M., Meier, M., Wirsing, M.: Which soft constraints do you prefer? In: Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA 2008). ENTCS, vol. 238(3), pp. 189–205. Elsevier, Amsterdam (2008)
10. Katoen, J.-P.: Advances in probabilistic model checking. In: Barthe, G., Hermenegildo, M.V. (eds.) VMCAI 2010. LNCS, vol. 5944, p. 25. Springer, Heidelberg (2009)
11. Kumar, N., Sen, K., Meseguer, J., Agha, G.: A rewriting based model for probabilistic distributed object systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 32–46. Springer, Heidelberg (2003)
12. Lanese, I., Montanari, U.: Synchronization algebras with mobility for graph transformations. In: Proceedings of the 3rd Joint Workshops on Foundations of Global Ubiquitous Computing (FGUC 2004). ENTCS, vol. 138(1), pp. 43–60. Elsevier, Amsterdam (2005)
13. Lluch Lafuente, A., Montanari, U.: Quantitative mu-calculus and CTL defined over constraint semirings. TCS 346(1), 135–160 (2005)
14. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. TCS 96(1), 73–155 (1992)
15. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. TCS 285(2), 359–405 (2002)
16. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming 60-61, 17–139 (2004)
17. Rote, G.: A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). Journal on Computing 34(3) (1985)
18. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education, London (2003)
19. Seidewitz, E.: What models mean. IEEE Journal on Software 20(5), 26–32 (2003)
20. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming 67(1-2), 226–293 (2006)

Reactive Semantics for Distributed UML Activities

Frank Alexander Kraemer and Peter Herrmann

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway
{kraemer,herrmann}@item.ntnu.no

Abstract. We define a *reactive* semantics for a subset of UML activities that is suitable as precise design language for reactive software systems. These semantics identify run-to-completion steps for execution on the level of UML activities as so-called activity steps. We show that activities adhering to these semantics and a set of rules lead to event-driven and bounded specifications that can be implemented automatically by model transformations and executed efficiently using runtime support systems.

Keywords: UML Activities, UML Semantics, Reactive Systems.

1 Introduction

UML 2.0 activities essentially denote in which order certain actions have to be executed to accomplish some task, and are therefore suitable for a wide range of applications. With their revision for the second major version of the UML standard [1], they were considerably enhanced with respect to supporting concurrent flows and hierarchically structured specifications. In the following, we focus on the application of UML activities on the domain of reactive systems. This class of systems, characterized by Pnueli as ones that *"maintain some interaction with their environment"* [2], is interesting, since with an increasing degree of connectivity of devices and the ubiquity of sensors that provide data, more and more applications fall into this category of systems. In general, these reactive applications and corresponding systems are characterized as follows:

- There is a high degree of concurrency in the applications, since typically several connections are simultaneously active and events from different sources can occur at any time.
- These applications are *event-driven*, that means they execute their behavior as reactions on events such as incoming signals from other devices, user interface interactions or updated sensor inputs.

These characteristics make the development of reactive systems quite demanding, especially with respect to concurrency. Achieving concurrency by processes executing in parallel is complicated, since synchronizations between them are difficult to understand and error-prone. In addition, the number of processes

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 17–31, 2010.
© IFIP International Federation for Information Processing 2010

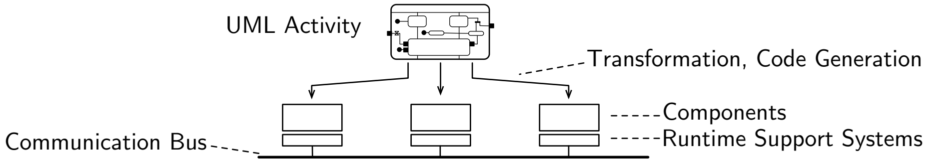


Fig. 1. Execution of components by runtime support systems

that can be executed efficiently in parallel is limited, in particular on mobile or embedded devices.

One way to deal with the complexity of concurrent executions is the introduction of runtime support systems [3], illustrated in the lower part of Fig. 1. These systems contain schedulers that control the execution of a component's behavior by dispatching events. Such events are the arrival of signals from other components, the expiration of timers, or internal signals that arise from local sensor data or interrupts. To be executable by a runtime support system, all application behavior must be expressed as transitions triggered by observable events as defined above. Moreover, communication must be implemented so that the sender is not blocked, for instance via an asynchronous message bus. These properties enable an execution with *run-to-completion* characteristics. The execution of a run-to-completion step is not preempted, and concludes with a stable state in which the runtime support system waits to dispatch the next observable event. This makes it relatively easy to execute also complex and highly concurrent behavior with very limited resources and a low number of system processes. Therefore, we are able to offer code generators different execution platforms, ranging from resource-constrained embedded systems on Sun SPOTs [4] to systems serving large numbers of users like Telenor's Connected Objects platform [5].

Our interest in activities is based on a number of characteristics they have that are relevant for the development of distributed, reactive systems: In comparison to state machines (that we have used before) they provide a high degree of concurrency by default, since activity flows execute independently from each other. We have also shown that activities can be grouped and abstracted in building blocks [6], which leads to system designs that are built to high proportions from reused solutions. Furthermore, since activities have the concept of partitions, they also describe collaborative behavior, and may encapsulate the behavior of several components that is necessary to fulfill a certain task, illustrated in Fig. 1. We therefore built a tool that takes UML activities and implements them automatically, using first a model transformation to UML components and state machines [7] and then a code generation step.

In order to use activities for reactive applications that are to be executed on an event-driven runtime support system as described above, their semantics need to address several issues:

– Activities should identify the run-to-completion steps executed by the runtime support system, as well as the observable event triggering a step.

– It must be clear on which location a run-to-completion step is performed, i.e., which component is responsible for executing it.
– Activities should make communication explicit, i.e., all necessary communication between components must be represented by activity flows.

One could argue, of course, that the abstraction level of activities does not need to deal with the same concepts as lower levels. For instance, non-local behavior of activities could be allowed, and communication patterns on the level of component execution could be synthesized automatically (as done for instance by Yamaguchi et al. [8]). However, we want the activities to express communication explicitly, for example to facilitate a security analysis as done in [9]. Furthermore, we want developers working on the level of activities to be still aware of costly operations such as sending to encourage efficient solutions. For the same reason we think it is beneficial to be aware of run-to-completion steps, since they form the temporal behavior of specifications.

In the following, we describe what we call *reactive* semantics for activities. Our intention is not to cover all details of the UML standard, but a subset of elements necessary to describe a wide range of reactive applications. We will present UML activities with an example and define some syntactic constraints in Sect. 3. Our semantics is based on run-to-completion steps, which on the activity level we call *activity steps*. These are defined in Sect. 4. The execution of activities based on activity steps is then described in Sect. 5. Section 6 discusses properties of activities and some additional constraints.

2 Related Work

There exists a variety of approaches that use different techniques to partially define and discuss semantics of UML 2.0 activities. Conrad Bock, one of the authors of the UML standard [1], covers the semantics of activities in a series of articles which informally clarify numerous semantic details. In particular, he describes the *traverse-to-completion* principle [10], according to which tokens pass only when all elements along a path accept the passing. This principle is implied by our semantics due to the definition of run-to-completion steps. Eshuis' work on model checking activity diagrams [11] defines their semantics by two mappings onto the model checker NuSMV, but in comparison to our work targets workflow systems, in which an activity is executed by a central workflow system that coordinates the execution of actions [12]. Störrle uses Coloured Petri Nets in [13] to define semantics of some UML activity elements. In [14], Störrle and Hausmann conclude that Petri Nets are probably not suitable to cover more advanced concepts, and point out the difficulty of combining all the various target domains. Barros and Gomez explain the semantics of actions with several input and output pins by "unfolding" these elements into activities with simpler control nodes [15]. Crane and Dingel [16] cover the detailed semantics of some specific UML action types, and describe in [17] a virtual machine for their interpretation. Engels et al. [18] define semantics of activities using Dynamic Meta Modeling (DMM), which is based on graph transformations. Sarstedt and

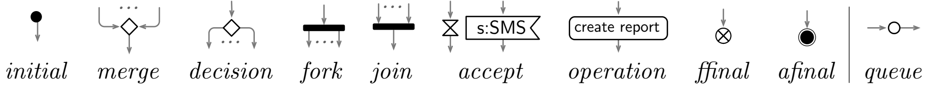


Fig. 2. Node kinds

Guttmann [19] use Abstract State Machines (ASMs) for the definitions of activity semantics. This work treats token passing on a very detailed level, removing some restrictions present in other approaches. To execute systems described by activities, they propose the interpretation of models [20].

In contrast to these approaches, we explicitly apply UML activities to the domain of reactive applications as characterized above, with an emphasis on execution mechanisms for runtime support systems. For this case, we found the semantics defined by the approaches mentioned above as not ideal, since they are either too general or target for example business processes, which pose different requirements on communication and synchronization. Therefore, we cannot use them as basis for the construction of executable components.

3 UML Activities

The UML standard uses simple text to explain activities, and characterizes their semantics as "Petri-like" [1, p. 324]. Some simple activity graphs can indeed easily mapped to Petri Nets. The mapping of more complex elements of UML activities, however, turns out to be difficult, especially for termination behavior, as pointed out by others [14,21,12]. In the following, we will therefore describe the reactive semantics of activities based on the intuitive token flows as found in Petri Nets, but use general state-transition systems in which also groups of tokens of several places can be removed simultaneously.

An activity is a directed graph A with a set of activity nodes V_A and a set of activity edges E_A. Function $kind_A \in [V_A \rightarrow K]$ assigns to each node a kind, with $K = \{initial, merge, decision, fork, join, accept, operation, afinal, ffinal, queue\}$, as illustrated in Fig. 2. Nodes of kind *accept* model accept event actions, which in our semantics represent either internal signal receptions or timer expirations. Internal signals are used to represent events from lower layers of a component, such as interrupts or events from user interfaces. Nodes of kind *operation* represent method calls. There are two kinds of final nodes, activity final nodes (*afinal*) that terminate an entire activity (hence removing tokens also from other places and preempting other behaviors), and flow final nodes (*ffinal*) that just consume tokens to conclude a flow.

An activity has a non-empty set of partitions P_A. In general, UML uses partitions to characterize commonalities among nodes. We use them strictly to denote different locations of executions. Thus, a flow crossing partition borders implies communication. For this communication, we assume an asynchronous message bus. This type of communication allows for an increased degree of concurrency,

but may introduce interleaving behavior that has to be taken care of by corresponding synchronization. Since this behavior is tightly interleaved with application logic, the delay of messages must be represented in the activity diagrams as well. For this reason, we introduce explicit queue places where activity flows cross partition borders. In Fig. 4, these are nodes q_1 to q_7.

Function $in_A \in [V_A \rightarrow 2^{E_A}]$ yields for each node its incoming edges, and function $out_A \in [V_A \rightarrow 2^{E_A}]$ its outgoing edges. Vice versa, we refer to $source_A(e)$ to get the source node of an edge, and $target_A(e)$ for its target. We assume that only merge and join nodes have more than one incoming edge, and only decision and fork nodes have more than one outgoing edge. All structural constraints are summarized by the rules in Fig. 3. Function $part_A \in [V_A \rightarrow P_A \cup (P_A \times P_A)]$ assigns partitions to nodes, so that each queue node is mapped to a pair of partitions modeling the transmission of signals between components (**PQ**). All other nodes are assigned to exactly one partition (**PN**). Rule **E1** ensures that edges do not have the same node as source and target, and **P1** ensures that there are no edges between two queues. These cases do not model useful behavior. Rules **P2** to **P4** ensure that edges do not cross partition borders without a queue node in between. (These rules are listed here for completeness. In practice, they

$$\textbf{IN1}\ \frac{v \in V_A \quad kind_A(v) \in \{initial\}}{|in_A(v)| = 0} \qquad \textbf{IN2}\ \frac{v \in V_A \quad kind_A(v) \in \{merge, join\}}{|in_A(v)| \geq 2}$$

$$\textbf{IN3}\ \frac{\begin{array}{c} v \in V_A \quad kind_A(v) \in \{decision, fork, \\ accept, operation, ffinal, afinal, queue\}\end{array}}{|in_A(v)| = 1} \qquad \textbf{OUT1}\ \frac{\begin{array}{c} v \in V_A \quad kind_A(v) \in \{initial, \\ merge, join, accept, operation\}\end{array}}{|out_A(v)| = 1}$$

$$\textbf{OUT2}\ \frac{\begin{array}{c} v \in V_A \\ kind_A(v) \in \{ffinal, afinal\}\end{array}}{|out_A(v)| = 0} \qquad \textbf{OUT3}\ \frac{\begin{array}{c} v \in V_A \\ kind_A(v) \in \{decision, fork\}\end{array}}{|out_A(v)| \geq 2}$$

$$\textbf{PN}\ \frac{v \in V_A \quad kind_A(v) \neq queue}{part_A(v) \in P_A} \qquad \textbf{PQ}\ \frac{q \in V_A \quad kind_A(q) = queue \quad p_1, p_2 \in P_A}{part_A(q) = \langle p_1, p_2 \rangle \quad p_1 \neq p_2}$$

$$\textbf{E1}\ \frac{e \in E_A}{source_A(e) \neq target_A(e)} \qquad \textbf{P1}\ \frac{e \in E_A \quad kind_A(source_A(e)) = queue}{kind_A(target_A(e)) \neq queue}$$

$$\textbf{P2}\ \frac{e \in E_A \quad kind_A(source_A(e)) \neq queue \quad kind_A(target_A(e)) \neq queue}{part_A(source_A(e)) = part_A(target_A(e))}$$

$$\textbf{P3}\ \frac{\begin{array}{c} e \in E_A \quad p, q \in P_A \\ kind_A(source_A(e)) \neq queue \\ kind_A(target_A(e)) = queue \\ part_A(target_A((e)) = \langle p, q \rangle \end{array}}{part_A(source(e)) = p} \qquad \textbf{P4}\ \frac{\begin{array}{c} e \in E_A \quad p, q \in P_A \\ kind_A(source_A(e)) = queue \\ kind_A(target_A(e)) \neq queue \\ part_A(source_A((e)) = \langle p, q \rangle \end{array}}{part_A(target(e)) = q}$$

Fig. 3. Rules for incoming and outgoing edges and partitions

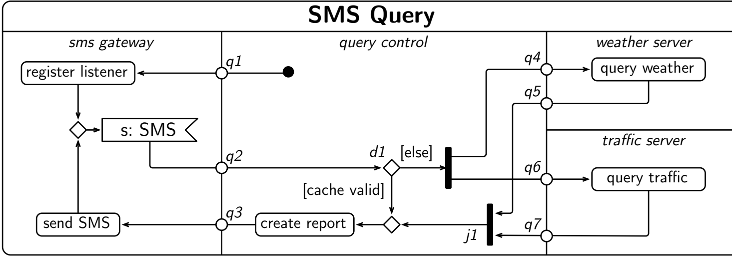


Fig. 4. Example activity for an SMS-based query system

are ensured by construction, since queue places are added automatically for edges crossing partitions).

Figure 4 shows a simplified SMS-based query service, in which customers can request weather and traffic information by SMS. The system consists of four components with different tasks, represented by separate activity partitions. The behavior is started by the query control server, which activates the SMS gateway to listen for incoming SMS messages. These are received via accept node *s: SMS*, emitting one token for each SMS that is forwarded to the query control server. There, a decision is made in *d1* whether the cache holds valid traffic and weather data. If yes, a report is created immediately. If data is not available locally, queries are sent towards the weather and traffic servers concurrently. Join *j1* then collects their responses, which are used to create the report that is forwarded to the SMS gateway and sent out.

4 Run-to-Completion Steps in Activities: Activity Steps

As explained in the introduction, the execution of components by runtime support systems is based on run-to-completion steps that are triggered by discrete, observable events. On the level of activities, a run-to-completion step is called *activity step*. With respect to the formal representation of an activity diagram, an activity step is a subgraph a with $V_a \subseteq V_A$ and $E_a \subseteq E_A$. An activity step covers all nodes and edges that describe behavior executed within one run-to-completion step. An activity diagram describes with its graph a set of activity steps. The complete set of activity steps for a diagram can be obtained by considering all possible subgraphs a_i of A, for which the rules in Fig. 5 hold.

- Whenever a node is part of an activity step, then at least one of its incoming or outgoing edges is part of the step as well (rule **V**). Vice-versa, for each edge part of a step, also its source and target nodes are part of the step (**E**).
- All initial nodes within the same partition release their tokens simultaneously, so that also all other initial nodes within the same partition are part of an activity step (**I**).

- For merge nodes, only one incoming edge is part of an activity step (**M**). This rules out intricate behavior in which two or more tokens pass the same edge within the same step, as discussed later.
- For decision nodes, several activity steps are produced. Each contains the incoming edge, the decision node, and exactly one outgoing edge (**D**). This means that an activity step executes exactly one branch of a decision.
- When a fork is part of an activity step, so are its incoming edge and all outgoing edges, since they are executed in parallel (**F**).
- When a join is part of an activity step, then there is at least one incoming edge part of the step as well (**J**).
- Operations are executed within one run-to-completion step, and thus the incoming and outgoing edge must be part of the same activity step (**O**).
- The sending and the reception at partition borders are part of separate activity steps. Therefore, for each queue place, either the incoming or the outgoing edge is part of the same activity step (**Q**).
- When a node of type *initial*, *accept* or *queue* is part of a step and its outgoing edge as well, then it is triggering the step. Rule **T2** ensures that each activity step has at least one such trigger, and rule **T1** ensures that is at most one.
- Accept nodes are covered by the general rules **V** and **E**.

Figure 6 shows the complete set of activity steps that can be produced from the activity in Fig. 4. Steps a_1 to a_7 are executed within the query control, steps a_9 to a_{11} by the SMS gateway, and a_8 and a_{12} by the weather resp. the traffic server. Some steps are especially interesting: Steps a_1 and a_5 model the arrival

$$\mathbf{V}\ \frac{v \in V_a}{in_A(v) \cap E_a \neq \emptyset \ \vee\ out_A(v) \cap E_a \neq \emptyset} \qquad \mathbf{E}\ \frac{e \in E_a}{source(e) \in V_a \quad target(e) \in V_a}$$

$$\mathbf{I}\ \frac{i \in V_a \quad part_A(i) = part_A(j) \quad kind_A(i) = kind_A(j) = initial}{j \in V_a}$$

$$\mathbf{M}\ \frac{m \in V_a \quad kind_A(m) = merge}{out_A(m) \subseteq E_a \quad |in_A(m) \cap E_a| = 1} \qquad \mathbf{D}\ \frac{d \in V_a \quad kind_A(d) = decision}{in_A(d) \subseteq E_a \quad |E_a \cap out_A(d)| = 1}$$

$$\mathbf{F}\ \frac{f \in V_a \quad kind_A(f) = fork}{in_A(f) \subseteq E_a \quad out_A(f) \subseteq E_a} \qquad \mathbf{J}\ \frac{j \in V_a \quad kind_A(j) = join}{in_A(j) \cap E_a \neq \emptyset}$$

$$\mathbf{O}\ \frac{o \in V_a \quad kind_A(o) = operation}{in_A(o) \subseteq E_a \quad out_A(o) \subseteq E_a} \qquad \mathbf{Q}\ \frac{q \in V_a \quad kind_A(q) = queue}{in_A(q) \cap E_a \neq \emptyset \Leftrightarrow out_A(q) \cap E_a = \emptyset}$$

$$\mathbf{T1}\ \frac{\begin{array}{c} t, u \in V_a \quad out_A(t) \cap E_a \neq \emptyset \quad part_A(t) = part_A(u) \\ kind_A(t) \in \{initial, accept, queue\} \quad kind_A(u) \in \{accept, queue\} \end{array}}{out_A(u) \cap E_a = \emptyset}$$

$$\mathbf{T2}\ \frac{\textsc{True}}{t \in V_a \quad out_A(t) \cap E_a \neq \emptyset \quad kind_A(t) \in \{initial, accept, queue\}}$$

Fig. 5. Rules for activity step subgraphs

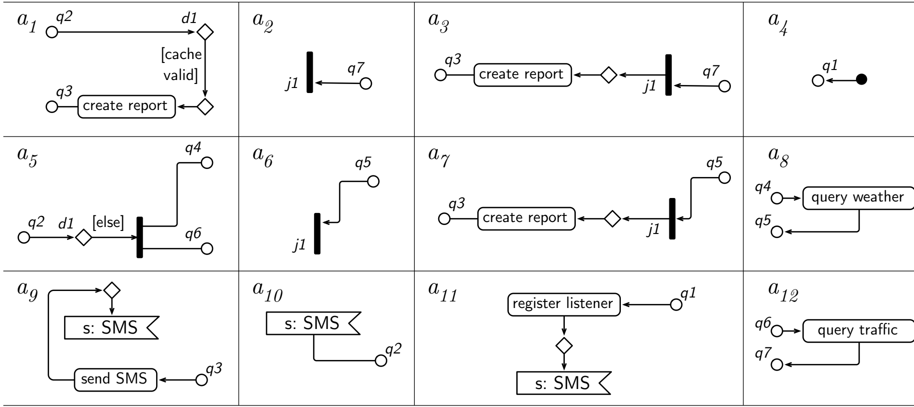


Fig. 6. Complete set of activity steps for the example

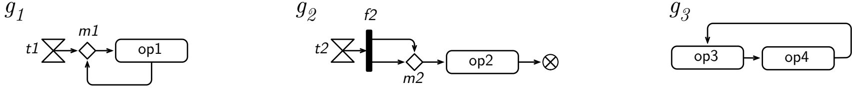


Fig. 7. Illegal subgraphs forbidden by rules

of an SMS by the main server from the SMS gateway. Together they cover the alternative branches introduced by decision node d_1, which either creates the report instantly or starts the query. Steps a_2 and a_3 represent the arrival of the results from the traffic server (via q_7). Step a_2 covers the situation that the weather information did not yet arrive (and therefore stops at j_2), while step a_3 can fire through j_1 since the weather information already arrived.

Figure 7 shows examples of subgraphs that are not valid activity steps. Subgraph g_1 results in infinite executions of *op1* and is therefore not desired. Subgraph g_2 executes *op2* twice. While this is not necessarily wrong, we think that this is probably not obvious to and by no means intended by developers, and should be forbidden. Subgraph g_3 shows behavior that simply is never reachable. The existence of such invalid subgraphs in a diagram are detected by rule contradictions. The subgraph g_1 is illegal since rule **O** and **E** would force that both incoming edges of *m1* are part of the activity step, which is forbidden by rule **M**. Similarly, in g_2 rule **F** would contradict rule **M**. Subgraph g_3 has no trigger, contradicting rule **T2**. The existence of rule contradictions signify inconsistent diagrams that have to be changed.

5 Execution Steps and Execution Semantics

Formally, the behavior of a reactive system A expressed by a UML activity is as a state transition system in which the states are represented as a placement of

tokens on the vertices and edges of the activity. For this reason, we define the type $tokens_A \triangleq 2^{[V_A \cup E_A \to \mathbb{N}]}$ of token mappings assigning each node and edge a natural number. Thus, a token mapping specifies a single system state.

A specific token mapping $init_A \in tokens_A$ refers to the token mapping of system A in its initial state. This initial mapping assigns one token to each of the vertices of kind *initial* while all the other nodes and edges are empty, as described by the following rules:

$$\textbf{INIT1}\ \frac{v \in V_A \quad kind_A(v) = initial}{init_A(v) = 1} \qquad \textbf{INIT2}\ \frac{v \in V_A \quad kind_A(v) \neq initial}{init_A(v) = 0}$$

$$\textbf{INIT3}\ \frac{e \in E_A}{init_A(e) = 0}$$

The state transitions are represented by execution steps which correspond to the activity steps and additional information describing the token settings before resp. after executing the step. Formally, we specify the execution step of an activity step represented by a subgraph $\langle V_a, E_a \rangle$ as the quadruple $ax_a = \langle V_a, E_a, pre_a, post_a \rangle$. $pre_a \in tokens_A$ describes the token setting of A before executing the activity step, and $post_a \in tokens_A$ denotes the token setting afterwards. For an ax_a to be valid, it must fulfill the following properties:

- In general, a token is added to accept and queue nodes when their incoming edge is part of an activity step (**IX**).
- In order to execute, the place representing a trigger must hold a token (**AX1**). If the trigger place is re-filled within the same step, its token count stays the same (**AX3**), otherwise it is reduced by one (**AX2**).
- For join nodes, two rules exist: Rule **JX1** models the firing of a join when tokens arrive at its incoming edges. All incoming edges not part of the activity step must already provide a token each. When the join fires, these tokens are deleted, and the step continues with the outgoing edge of the join. Rule **JX2** handles the arrival of tokens at a join when it is not yet complete. As consequence, it adds these arriving tokens but does not continue.
- Once a final node is part of an activity step, all tokens within the same partition are removed (see rule **AFX1** for tokens on edges and **AFX2** for tokens on nodes). The latter rule also removes all tokens within queues *towards* the partition containing the activity final node. This takes into account that the component implementing the terminated partition is switched off, and any remaining signals towards it are discarded. The additional precondition

  $$nofinal(n\colon 2^{N_A}, p\colon P_A) \triangleq \forall f\colon f \in n \wedge part_A(f) = p \Rightarrow kind_A(f) \neq afinal$$

  added to rules **IX**, **AX3**, **JX1** and **JX2** ensures that these rules only apply if no activity final node is part of the activity step, to give rules **AFX1** and **AFX2** priority.
- All vertices and edges of A not mentioned by one of the rules explicitly do not change their token setting in the execution step ax_a.

$$\textbf{IX}\ \frac{v \in V_a \quad kind_A(v) \in \{accept, queue\} \quad in_A(v) \cap E_a \neq \emptyset \quad out_A(v) \cap E_a = \emptyset \quad nofinal(V_a, part(v))}{post_a(v) = pre_a(v) + 1}$$

$$\textbf{AX1}\ \frac{v \in V_a \quad kind_A(v) \in \{initial, accept, queue\} \quad out_A(v) \cap E_a \neq \emptyset}{pre_a(v) > 0} \qquad \textbf{AX2}\ \frac{v \in V_a \quad kind_A(v) \in \{initial, accept, queue\} \quad out_A(v) \cap E_a \neq \emptyset \quad in_A(v) \cap E_a = \emptyset}{post_a(v) = pre_a(v) - 1}$$

$$\textbf{AX3}\ \frac{v \in V_a \quad kind_A(v) = accept \quad in_A(v) \cap E_a \neq \emptyset \quad out_A(v) \cap E_a \neq \emptyset \quad nofinal(V_a, part(v))}{post_a(v) = pre_a(v)}$$

$$\textbf{JX1}\ \frac{v \in V_a \quad kind_A(v) = join \quad es = in_A(v) \cap E_a \quad \forall e \in in_A(v) \setminus es : pre_a(e) > 0 \quad nofinal(V_a, part(v))}{out_A(v) \subseteq E_a \quad \forall e \in in_A(v) \setminus es : post_a(e) = pre_a(e) - 1}$$

$$\textbf{JX2}\ \frac{v \in V_a \quad kind_A(v) = join \quad es = in_A(v) \cap E_a \quad \exists e \in in_A(v) \setminus es : pre_a(e) = 0 \quad nofinal(V_a, part(v))}{out_A(v) \cap E_a = \emptyset \quad \forall e \in es : post_a(e) = pre_a(e) + 1}$$

$$\textbf{AFX1}\ \frac{e \in E_A \quad f \in V_a \quad kind_A(f) = afinal \quad part(f) = part(target_A(e))}{post_a(e) = 0} \qquad \textbf{AFX2}\ \frac{f \in V_a \quad v \in V_A \quad kind_A(f) = afinal \quad part(v) = part(f) \vee part(v) = \langle _, part(f) \rangle}{post_a(v) = 0}$$

Fig. 8. Execution rules for activity steps

We define the set AX_A as the set of execution steps ax_a each following the rules mentioned above. This set contains execution steps that are not reachable. We define the set of reachable execution steps $RAX_A \subseteq AX_A$ by means of the following two rules:

$$\textbf{R1}\ \frac{a \in AX_A \quad \forall v \in V_A : kind_A(v) = initial \Leftrightarrow pre_a(v) = 1 \quad \forall v \in V_A : kind_A(v) \neq initial \Leftrightarrow pre_a(v) = 0 \quad \forall e \in E_A : pre_a(e) = 0}{a \in RAX_A}$$

$$\textbf{R2}\ \frac{a, b \in AX_A \quad \forall v \in V_A : pre_a(v) = post_b(v) \quad \forall e \in E_A : pre_a(e) = post_b(e) \quad b \in RAX_A}{a \in RAX_A}$$

The rules define recursively that the reachable execution steps a are those containing a token setting pre_a reachable by a finite trace of execution steps from the initial token setting. The behavior of system Sys_A is then defined by the state transition system expressed by the quadruple $Sys_A \triangleq \langle V_A, E_A, init_A, RAX_A \rangle$.

6 Properties of Activities with Reactive Semantics

In the following, we will discuss the properties of activities with reactive semantics. The properties in Sect. 6.2 and 6.3 require additional behavioral invariants, that is, they only hold for a subset of activities. We will assure these by additional rules that have to hold for any $a \in RAX_A$. In practice, these rules are verified by model checking, as discussed in [22].

6.1 Event-Driven Execution and Run-to-Completion

As described in the introduction, the events observable by a runtime support system are the expiration of timers, the arrival of internal signals, or the arrival of signals from other components. On the activity level, these events are modeled by accept event actions and activity flows that cross partition borders via queues. The initial startup of a component is also an event, modeled by initial nodes in activities. Due to rules **T1** and **T2**, we ensure that each execution step declares exactly one such trigger, meaning that activity steps started by an observable event. Furthermore, tokens may be only placed according to two properties:

(i) Tokens may rest on a vertex only if it is of kind *initial*, *accept*, or *queue*.
(ii) Tokens may rest on edges if they lead to a join node, but only if the join is not yet complete, i.e., there is one incoming edge that cannot offer a token.

Property (i) ensures that tokens only wait in places that imply waiting for an observable event. In initial nodes, this is the start of the system, in queues the arrival of the signal and for accept nodes the expiration of a timer or the arrival of an internal signal, resp. Property (ii) is more subtle. Tokens may wait on edges preceding join nodes. But since join nodes do not declare any observable events, tokens may only rest before a join if the join is not yet complete. The join fires through within the same step once the last missing token arrives. Hence, a token stored in an incomplete join implies waiting for another observable event. Formally, (i) and (ii) can be expressed as $P \triangleq E_1 \wedge E_2 \wedge E_3 \wedge E_4$, with

$$E_1 \triangleq \forall v \in V_A : \mathit{kind}(v) \in \{\mathit{initial}, \mathit{accept}, \mathit{queue}\} \vee \mathit{init}_A(v) = 0$$

$$E_2 \triangleq \forall e \in E_A : \mathit{kind}(\mathit{target}_A(e)) \neq \mathit{join} \wedge \mathit{init}_A(e) = 0 \\ \vee\ \exists f \in E_A : f \in \mathit{in}(\mathit{target}_A(e)) \wedge \mathit{init}_A(f) = 0$$

$$E_3 \triangleq \forall v \in V_A \forall a \in RAX_A : \mathit{kind}(v) \in \{\mathit{initial}, \mathit{accept}, \mathit{queue}\} \vee \mathit{post}_a(v) = 0$$

$$E_4 \triangleq \forall e \in E_A \forall a \in RAX_A : \mathit{kind}(\mathit{target}_A(e)) \neq \mathit{join} \wedge \mathit{post}_a(e) = 0 \\ \vee\ \exists f \in E_A : f \in \mathit{in}(\mathit{target}_A(e)) \wedge \mathit{post}_a(f) = 0$$

E_1 and E_2 describe that (i) and (ii) hold in the initial state of the system while E_3 and E_4 guarantee that all execution steps $a \in RAX_A$ preserve them as well. E_1 holds due to rules **INIT1** and **INIT2** and E_2 holds due to rule **INIT3**.

To prove E_3, we consider the rules in Fig. 8. The only rule describing that the token setting of a vertex v after executing an execution step a can be greater

than before (i.e., $post_a(v) > pre_a(v)$) is **IX**. But according to the second premise of the rule, the token setting may only be increased for tokens of kind *accept* and *queue*. Thus, if a vertex of a type other than *initial*, *accept* or *queue* has no token placed on it before the execution of the execution step, it will carry none afterwards as well. In consequence, it will never carry a token at all.

The proof of E_4 is quite similar. The only rule in Fig. 8 modeling an increase of a token setting on an edge in an execution step is **JX2**. Yet it does not allow the placement of new tokens on edges leading to a vertex not being of kind *join* as expressed by the third premise. So, the first disjunct of E_4 holds and in all system states only edges into a join may have tokens at all. Further, the forth premise of the rule states that there is an edge leading into the join node to which no token is assigned. As this edge is not an element of the edges receiving new tokens (expressed by set es), it will not carry a token after firing the execution step. Thus, the second disjunct of E_4 holds as well and a join node will always have an incoming edge without a token placed on it.

6.2 Realizability and Distribution

To be executable as one run-to-completion step, an activity step must only have direct local effects, and only depend on data that is available locally. Rules **Q** and **P1** to **P4** split up activity steps at partition borders. Within one activity step, all nodes except queues are therefore part of the same partition. For flows crossing partition borders, we take the unavoidable communication delay into account by explicit queue nodes, so that they can be implemented using some communication middleware. For some nodes, the general semantics of UML describe also non-local dependencies, that motivate some further constraints:

- As an extension to standard UML, we assume that variables, like activity nodes, are assigned to partitions. Guards and actions are only allowed to access variables within their own partition.
- Initial nodes are started simultaneously, but only those within the same partition. This is already ensured by rule **I**.
- In standard UML, accept event actions without incoming edge denote that they are activated with the surrounding activity. Instead, we require these actions to have an incoming edge (rule **IN3**), to model activation explicitly.
- Activity final nodes terminate in standard UML the entire activity. With the reactive semantics, they only remove tokens within the *same* partition (rules **AFX1** and **AFX2**). Since messages towards a terminated partition are discarded, queues towards a terminated partition are emptied as well.

To comply with the general semantics of activity nodes in which an activity final node terminates the behavior in *all* partitions, the other partitions must not be able to execute any further activity steps. This is ensured by the additional rules **TRM1** and **TRM2**. The former states that whenever an activity final node is reached, there must not be any token in accept nodes of other partitions. The latter states that all queues not targeting the terminated partition must be empty as well. Since tokens offered to join nodes cannot trigger any behavior on their own, they do not have to be removed upon termination.

$$\textbf{TRM1}\ \frac{f \in V_a \quad v \in V_A \quad kind_A(f) = afinal \quad kind_A(v) = accept \quad part_A(v) \neq part_A(f)}{pre_a(v) = post_a(v) = 0}$$

$$\textbf{TRM2}\ \frac{f \in V_a \quad q \in V_A \quad kind_A(f) = afinal \quad kind_A(q) = queue \quad part_A(q) \neq \langle _, f \rangle}{pre_a(q) = post_a(q) = 0}$$

$$\textbf{AB}\ \frac{v \in V_a \quad kind_A(v) = accept}{post_a(v) \leq 1} \qquad \textbf{JB}\ \frac{v \in V_a \quad kind_A(v) = join \quad e \in in_A(v)}{post_a(e) \leq 1}$$

$$\textbf{QB}\ \frac{v \in V_a \quad kind_A(v) = queue}{post_a(v) \leq maxqueue}$$

Fig. 9. Additional rules for activities

6.3 Boundedness

Since the number of places needed to describe an activity is limited, the state space implied by a specification is finite if (and only if) each place only contains a bounded number of tokens, i.e., $|tokens_A(x) < N|$, with N as a finite boundary. Rules **IX** and **JX2** are the only rules increasing token places.

- Accept nodes are either enabled or disabled, represented by one or zero tokens on their corresponding place. We found that adding more than one token in an accept node is in most cases unintended and an indicator of a design flaw. We therefore rule out such behavior by rule **AB**.
- Places before join nodes can hold many tokens which implies buffering of data or control flow. We found that this makes activities harder to understand, without adding any expressiveness for reactive systems; we rather recommend to use explicit building blocks to describe buffering and rule out behaviors in which tokens accumulate before joins by rule **JB**.
- Queues between partitions need to be bounded as well. That means, they must not exceed a certain value $maxqueue \in \mathbb{N}$, expressed by rule **QB**.

If the boundedness rules hold, the set of reachable execution steps RAX_A will be finite as it is lower or equal to the product of run-to-completion steps times possible token markings following the boundedness constraints.

7 Concluding Remarks

We described a reactive semantics for UML activities, in which each execution step is triggered by an observable event. This is motivated by existing mechanisms present in runtime support systems for efficient but nevertheless simple scheduling and execution of highly concurrent behavior. As a consequence of the reactive semantics, the components produced by our model transformation [7] from activities have the same efficiency as if they would have been produced manually by an experienced designer, i.e., do not contain any overhead for token control, and it is not visible that they were generated from UML activities.

We have implemented comprehensive tool support with Arctis [22], a set of Eclipse plug-ins. It enables editing, analyzing and automatically implementing activity-based specifications. The syntactical rules from Fig. 3 are ensured by the editor, highlighting erroneous parts of the graph. Similarly, the tool can produce all activity steps implied by a diagram, using the rules from Fig. 5. Rule contradictions that signify illegal diagrams are detected and explained to the user. Finally, the additional rules for sound behavior in Fig. 9 are verified by model checking [22]. Our tool also supports data in activities, which we did not treat here. Our experience from implementing data shows that their semantics can be covered by extending the semantics for control flows in a straight-forward way. Operations with more than one incoming flow, for instance, can be modeled similar to join nodes. More advanced nodes can be modeled by dedicated building blocks. Concerning the expressiveness of the chosen reactive semantics we point to numerous case studies (summarized in [6]) that exemplify their application in various domains. With an abstraction mechanism described in [6] such solutions can also be encapsulated by dedicated building blocks from which large system designs can be produced in a scalable manner.

References

1. Object Management Group: Unified Modeling Language: Superstructure, version 2.2, formal/2009-02-02 (2009)
2. Pnueli, A.: Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In: Rozenberg, G., de Bakker, J.W., de Roever, W.-P. (eds.) Current Trends in Concurrency. LNCS, vol. 224, pp. 510–584. Springer, Heidelberg (1986)
3. Bræk, R., Haugen, Ø.: Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. Prentice Hall, Englewood Cliffs (1993)
4. Kraemer, F.A., Slåtten, V., Herrmann, P.: Model-Driven Construction of Embedded Applications based on Reusable Building Blocks – An Example. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 1–18. Springer, Heidelberg (2009)
5. Herstad, A., Nersveen, E., Samset, H., Storsveen, A., Svaet, S., Husa, K.E.: Connected Objects: Building a Service Platform for M2M. In: Beyond the Bit Pipe. Proccedings of the 13th ICIN Conference (2009)
6. Kraemer, F.A., Herrmann, P.: Automated Encapsulation of UML Activities for Incremental Development and Verification. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 571–585. Springer, Heidelberg (2009)
7. Kraemer, F.A., Herrmann, P.: Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In: Ehring, K., Giese, H. (eds.) 6th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT), Proceedings. Electronic Communications of the EASST, vol. 7. EASST (2007)
8. Yamaguchi, H., El-Fakih, K., von Bochmann, G., Higashino, T.: Protocol Synthesis and Re-Synthesis with Optimal Allocation of Resources based on Extended Petri Nets. Distrib. Comput. 16(1), 21–35 (2003)
9. Gunawan, L.A., Herrmann, P., Kraemer, F.A.: Towards the Integration of Security Aspects into System Development using Collaboration-Oriented Models. In: Security Technology. International Conference on Security Technology (SecTech 2009), Proceedings. CCIS, vol. 58, pp. 72–85. Springer, Heidelberg (2009)

10. Bock, C.: UML 2 Activity and Action Models, Part 4: Object Nodes. Journal of Object Technology 3(1), 27–41 (2004)
11. Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. ACM Transactions on Software Engineering and Methodology 15(1), 1–38 (2006)
12. Eshuis, R., Wieringa, R.: Comparing Petri Net and Activity Diagram Variants for Workflow Modelling - A Quest for Reactive Petri Nets. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 321–351. Springer, Heidelberg (2003)
13. Störrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. Electronic Notes in Theoretical Computer Science 127, 35–52 (2005)
14. Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities. In: Liggesmeyer, P., Pohl, K., Goedicke, M. (eds.) Software Engineering, Fachtagung des GI-Fachbereichs Softwaretechnik. LNI, vol. 64, pp. 117–128. GI (2005)
15. Barros, J.P., Gomes, L.: Actions as Activities and Activities as Petri Nets. In: Workshop on Critical Systems Development with UML, Proceedings (2003)
16. Crane, M.L., Dingel, J.: Towards a Formal Account of a Foundational Subset for Executable UML Models. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 675–689. Springer, Heidelberg (2008)
17. Crane, M.L., Dingel, J.: Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities. In: CASCON 2008: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research, pp. 96–110. ACM, New York (2008)
18. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities Using Dynamic Meta Modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 76–90. Springer, Heidelberg (2007)
19. Sarstedt, S., Guttmann, W.: An ASM Semantics of Token Flow in UML 2 Activity Diagrams. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 349–362. Springer, Heidelberg (2007)
20. Sarstedt, S., Gessenharter, S., Kohlmeyer, J., Raschke, A., Schneiderhan, M.: ActiveChartsIDE: An Integrated Software Development Environment Comprising a Component for Simulating UML 2 Activity Charts. In: European Simulation and Modelling Conference (ESM 2005), Proceedings, pp. 66–73 (2005)
21. van der Aalst, W., Hofstede, T.: Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In: Jensen, K. (ed.) Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002), Proceedings. DAIMI, vol. 560, pp. 1–20 (2002)
22. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software 82(12), 2068–2080 (2009)

Statistical Abstraction and Model-Checking of Large Heterogeneous Systems*

Ananda Basu[1], Saddek Bensalem[1], Marius Bozga[1], Benoît Caillaud[2], Benoît Delahaye[3], and Axel Legay[2]

[1] Verimag Laboratory, Université Joseph Fourier Grenoble, CNRS
[2] INRIA/IRISA, Rennes, France
[3] Université de Rennes 1/IRISA, Rennes, France

Abstract. We propose a new simulation-based technique for verifying applications running within a large heterogeneous system. Our technique starts by performing simulations of the system in order to learn the context in which the application is used. Then, it creates a stochastic abstraction for the application, which takes the context information into account. This smaller model can be verified using efficient techniques such as statistical model checking. We have applied our technique to an industrial case study: the cabin communication system of an airplane. We use the BIP toolset to model and simulate the system. We have conducted experiments to verify the clock synchronization protocol i.e., the application used to synchronize the clocks of all computing devices within the system.

1 Introduction

Systems integrating multiple heterogeneous distributed applications communicating over a shared network are typical in various sensitive domains such as aeronautic or automotive embedded systems. Verifying the correctness of a particular application inside such a system is known to be a challenging task, which is often beyond the scope of existing exhaustive validation techniques. The main difficulty comes from network communication which makes all applications interfering and therefore forces exploration of the full state-space of the system.

Statistical Model Checking [8,13,15] has recently been proposed as an alternative to avoid an exhaustive exploration of the state-space of the model. The core idea of the approach is to conduct some simulations of the system and then use statistical results in order to decide whether the system satisfies the property or not. Statistical model checking techniques can also be used to estimate the probability that a system satisfies a given property [8,7]. Of course, in contrast with an exhaustive approach, a simulation-based solution does not guarantee a correct result. However, it is possible to bound the probability of making an error. Simulation-based methods are known to be far less memory and time intensive than exhaustive ones, and are sometimes the only option [16,10]. Statistical model checking is widely accepted in various research areas such as systems

* This work has been supported by the Combest EU project.

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 32–46, 2010.
© IFIP International Federation for Information Processing 2010

biology [6,11] or software engineering, in particular for industrial applications. There are several reasons for this success. First, it is very simple to implement, understand and use. Second, it does not require extra modeling or specification effort, but simply an operational model of the system, that can be simulated and checked against state-based properties. Third, it allows model-checking of properties [5] that cannot be expressed in classical temporal logics. Nevertheless, statistical-model checking still suffers from the system's complexity. In particular, for the case of heterogeneous systems, the number of components and their interactions are limiting factors on the number and length of simulations that can be conducted and hence on the accuracy of the statistical estimates.

We propose to exploit the structure of the system in order to increase the efficiency of the verification process. The idea is conceptually simple: instead of performing an analysis of the entire system, we propose to analyze each application separately, but under some particular context/execution environment. This context is a *stochastic abstraction* that represents the interactions with other applications running within the system and sharing the computation and communication resources. We propose to build such a context automatically by simulating the system and learning the probability distributions of key characteristics impacting the functionality of the given application.

The overall contribution of this paper is an application of the above method on an industrial case study, the *heterogeneous communication system* (HCS for short) deployed for cabin communication in a civil airplane. HCS is a heterogeneous system providing entertainment services (e.g., audio/video on passengers demand) as well as administrative services (e.g., cabin illumination, control, audio announcements), which are implemented as distributed applications running in parallel, across various devices within the plane and communicating through a common Ethernet-based network. The HCS system has to guarantee stringent requirements, such as reliable data transmission, fault tolerance, timing and synchronization constraints. An important requirement, which will be studied in this paper, is the *accuracy of clock synchronization* between different devices. This latter property states that the difference between the clocks of any two devices should be bounded by a small constant, which is provided by the user and depends on his needs. Hence, one must be capable of computing the smallest bound for which synchronization occurs and compare it with the bound expected by the user. Unfortunately, due to the large number of heterogeneous components that constitute the system, deriving such a bound manually from the textual specification is an unfeasible task. In this paper, we propose a formal approach that consists in building a formal model of the HCS, then applying simulation-based algorithms to this model in order to deduce the smallest value of the bound for which synchronization occurs. We start with a fixed value of the bound and check whether synchronization occurs. If yes, then we make sure that this is the best one. If no, we restart the experiment with a new value.

At the top of our approach, there should be a tool that is capable of modeling heterogeneous systems as well as simulating their executions and the interactions between components. In this paper, we propose to use the BIP toolset [2]

for doing so. BIP (*Behaviour-Interaction-Priority*) supports a methodology for building systems from atomic components encapsulating behavior, that communicate through interactions, and coordinate through priorities. BIP also offers a powerful engine to simulate the system and can thus be combined with a statistical model checking algorithm in order to verify properties. Our first contribution is to study all the requirements for the HCS to work properly and then derive a model in BIP. Our second contribution is to study the accuracy of clock synchronization between several devices of the HCS. In HCS the clock synchronization is ensured by the *Precision Time Protocol* (PTP for short) [1], and the challenge is to guarantee that PTP maintains the difference between a master clock (running on a designated server within the system) and all the slave clocks (running on other devices) under some bound. Since this bound cannot be pre-computed, we have to verify the system for various values of the bound until we find a suitable one. Unfortunately, the full system is too big to be analyzed with classical exhaustive verification techniques. A solution could be to remove all the information that is not related to the devices under consideration. This is in fact not correct as the behavior of the PTP protocol is influenced by the other applications running in parallel within the heterogeneous system. Our solution to this state-space explosion problem is in two steps (1) we will build a stochastic abstraction for a part of the PTP application between the server and a given device; the stochastic part will be used to model the general context in which PTP is used, (2) we will apply statistical model checking on the resulting model.

Thanks to this approach, we have been able to derive precise bounds that guarantee proper synchronization for all the devices of the system. We also computed the probability of satisfying the property for smaller values of the bound, i.e., bounds that do not satisfy the synchronization property with probability 1. Being able to provide such information is of clear importance, especially when the best bound is too high with respect to the user's requirements. We have observed that the values we obtained strongly depend on the position of the device in the network. We also estimated the average and worst proportion of failures per simulation for bounds that are smaller than the one that guarantees synchronization. Checking this latter property has been made easy because BIP allows us to reason on one execution at a time. Finally, we have also considered the influence of clock drift on the synchronisation results. The experiments highlight the generality of our technique, which could be applied to other versions of the HCS as well as to other heterogeneous applications.

Due to space limitations, several constructions and algorithms are given in a technical report [3].

2 An Overview of Statistical Model Checking

Consider a stochastic system $\mathcal{S}$ and a property ϕ. *Statistical model checking* refers to a series of simulation-based techniques that can be used to answer two questions: (1) **Qualitative:** Is the probability that $\mathcal{S}$ satisfies ϕ greater or equal

to a certain threshold? and (2) **Quantitative:** What is the probability that $\mathcal{S}$ satisfies ϕ? Contrary to numerical approaches, the answer is given up to some correctness precision. In the rest of the section, we overview several statistical model checking techniques. Let B_i be a discrete random variable with a Bernoulli distribution of parameter p. Such a variable can only take 2 values 0 and 1 with $Pr[B_i = 1] = p$ and $Pr[B_i = 0] = 1 - p$. In our context, each variable B_i is associated with one simulation of the system. The outcome for B_i, denoted b_i, is 1 if the simulation satisfies ϕ and 0 otherwise.

2.1 Qualitative Answer Using Statistical Model Checking

The main approaches [15,13] proposed to answer the qualitative question are based on *hypothesis testing*. Let $p = Pr(\phi)$, to determine whether $p \geq \theta$, we can test $H : p \geq \theta$ against $K : p < \theta$. A test-based solution does not guarantee a correct result but it is possible to bound the probability of making an error. The *strength* (α, β) of a test is determined by two parameters, α and β, such that the probability of accepting K (respectively, H) when H (respectively, K) holds, called a Type-I error (respectively, a Type-II error) is less or equal to α (respectively, β). A test has *ideal performance* if the probability of the Type-I error (respectively, Type-II error) is exactly α (respectively, β). However, these requirements make it impossible to ensure a low probability for both types of errors simultaneously (see [15] for details). A solution is to use an *indifference region* $[p_1, p_0]$ (with θ in $[p_1, p_0]$) and to test $H_0 : p \geq p_0$ against $H_1 : p \leq p_1$. We now sketch two hypothesis testing algorithms.

Single Sampling Plan. To test H_0 against H_1, we specify a constant c. If $\sum_{i=1}^{n} b_i$ is larger than c, then H_0 is accepted, else H_1 is accepted. The difficult part in this approach is to find values for the pair (n, c), called a *single sampling plan (SSP in short)*, such that the two error bounds α and β are respected. In practice, one tries to work with the smallest value of n possible so as to minimize the number of simulations performed. Clearly, this number has to be greater if α and β are smaller but also if the size of the indifference region is smaller. This results in an optimization problem, which generally does not have a closed-form solution except for a few special cases [15]. In his thesis [15], Younes proposes a binary search based algorithm that, given p_0, p_1, α, β, computes an approximation of the minimal value for c and n.

Sequential probability ratio test. The sample size for a single sampling plan is fixed in advance and independent of the observations that are made. However, taking those observations into account can increase the performance of the test. As an example, if we use a single plan (n, c) and the $m > c$ first simulations satisfy the property, then we could (depending on the error bounds) accept H_0 without observing the $n - m$ other simulations. To overcome this problem, one can use the *sequential probability ratio test (SPRT in short)* proposed by Wald [14]. The approach is briefly described below.

In SPRT, one has to choose two values A and B $(A > B)$ that ensure that the strength of the test is respected. Let m be the number of observations that have been made so far. The test is based on the following quotient:

$$\frac{p_{1m}}{p_{0m}} = \prod_{i=1}^{m} \frac{Pr(B_i = b_i \mid p = p_1)}{Pr(B_i = b_i \mid p = p_0)} = \frac{p_1^{d_m}(1-p_1)^{m-d_m}}{p_0^{d_m}(1-p_0)^{m-d_m}}, \tag{1}$$

where $d_m = \sum_{i=1}^{m} b_i$. The idea behind the test is to accept H_0 if $\frac{p_{1m}}{p_{0m}} \geq A$, and H_1 if $\frac{p_{1m}}{p_{0m}} \leq B$. The SPRT algorithm computes $\frac{p_{1m}}{p_{0m}}$ for successive values of m until either H_0 or H_1 is satisfied; the algorithm terminates with probability 1[14]. This has the advantage of minimizing the number of simulations. In his thesis [15], Younes proposed a logarithmic based algorithm SPRT that given p_0, p_1, α and β implements the sequential ratio testing procedure.

2.2 Quantitative Answer Using Statistical Model Checking

In [8,12] Peyronnet et al. propose an estimation procedure to compute the probability p for $\mathcal{S}$ to satisfy ϕ. Given a *precision* δ, Peyronnet's procedure, which we call PESTIMATION, computes a value for p' such that $|p' - p| \leq \delta$ with *confidence* $1 - \alpha$. The procedure is based on the *Chernoff-Hoeffding bound* [9]. Let $B_1 \ldots B_m$ be m discrete random variables with a Bernoulli distribution of parameter p associated with m simulations of the system. Recall that the outcome for each of the B_i, denoted b_i, is 1 if the simulation satisfies ϕ and 0 otherwise. Let $p' = (\sum_{i=1}^{m} b_i)/m$, then Chernoff-Hoeffding bound [9] gives $Pr(|p' - p| > \delta) < 2e^{-\frac{m\delta^2}{4}}$. As a consequence, if we take $m \geq \frac{4}{\delta^2}\log(\frac{2}{\alpha})$, then $Pr(|p' - p| \leq \delta) \geq 1 - \alpha$. Observe that if the value p' returned by PESTIMATION is such that $p' \geq \theta - \delta$, then $\mathcal{S} \models Pr_{\geq\theta}$ with confidence $1 - \alpha$.

2.3 Playing with Statistical Model Checking Algorithms

The efficiency of the above algorithms is characterized by the number of simulations needed to obtain an answer. This number may change from executions to executions and can only be estimated (see [15] for an explanation). However, some generalities are known. For the qualitative case, it is known that, except for some situations, SPRT is always faster than SSP. When $\theta = 1$ (resp. $\theta = 0$) SPRT degenerates to SSP; this is not problematic since SSP is known to be optimal for such values. PESTIMATION can also be used to solve the qualitative problem, but it is always slower than SSP [15]. If θ is unknown, then a good strategy is to estimate it using PESTIMATION with a low confidence and then validate the result with SPRT and a strong confidence.

3 Validation Method and the BIP Toolset

Consider a system consisting of a set of distributed applications running on several computers and exchanging messages on a shared network infrastructure.

Assume also that network communication is subject to given bandwidth restrictions as well as to routing and scheduling policies applied on network elements. Our method attempts to reduce the complexity of validation of a particular application of such system by decoupling the timing analysis of the network and functional analysis of each application.

We start by constructing a model of the whole system. This model must be executable, i.e., it should be possible to obtain execution traces, annotated with timing information. For a chosen application, we then learn the probability distribution laws of its message delays by simulating the entire system. The method then constructs a reduced stochastic model by combining the application model where the delays are defined according to the laws identified at the previous step. Finally, the method applies statistical model-checking on the resulting stochastic model.

Our models are specified within the BIP framework [2]. BIP is a component-based framework for construction, implementation and analysis of systems composed of heterogeneous components. In particular, BIP fulfills all the requirements of the method suggested above, that are, models constructed in BIP are operational and can be thoroughly simulated. BIP models can easily integrate timing constraints, which are represented with discrete clocks. Probabilistic behaviour can also be added by using external C functions.

The BIP framework is implemented within the BIP toolset [4], which includes a rich set of tools for modeling, execution, analysis (both static and on-the-fly) and static transformations of BIP models. It provides a dedicated programming language for describing BIP models. The front-end tools allow editing and parsing of BIP programs, and generating an intermediate model, followed by code generation (in C) for execution and analysis on a dedicated middleware platform. The platform also offers connections to external analysis tools. A more complete description of BIP can be found in [3].

4 Case Study: Heterogeneous Communication System

The case study concerns a distributed heterogeneous communication system (HCS) providing an all electronic communication infrastructure to be deployed, typically for cabin communication in airplanes or for building automation. The HCS system contains various devices such as sensors (video camera, smoke detector, temperature, pressure, etc.) and actuators (loudspeakers, light switches, temperature control, signs, etc.) connected through a wired communication network to a common server. The server runs a set of services to monitor the sensors and to control the actuators. The devices are connected to the server using network access controllers (NAC) as shown for an example architecture in Figure 1. An extended star-like HCS architecture with several daisy chains of NACs and devices is presented in [3].

The system-wide HCS architecture is highly heterogeneous. It includes hardware components and software applications ensuring functions with different characteristics and degree of criticality e.g., audio streaming, device clock synchronisation, sensor monitoring, video surveillance. It also integrates different

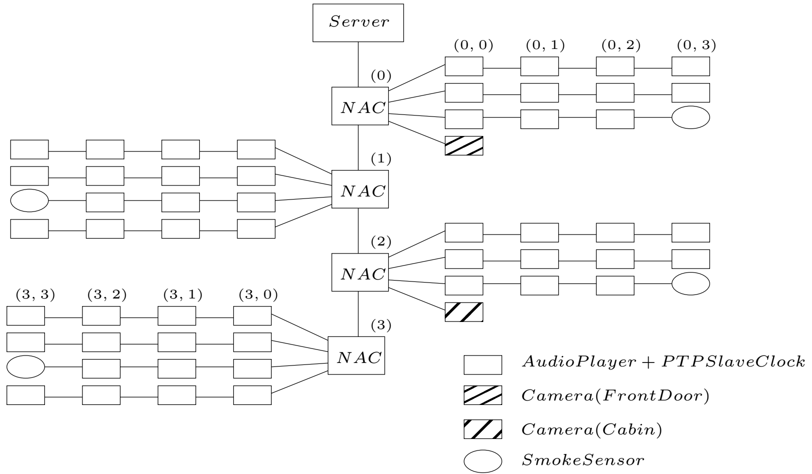


Fig. 1. HCS Example Model

communication and management protocols between components. The HCS system has to guarantee stringent requirements, such as reliable data transmission, fault tolerance, timings and synchronization constraints. For example, the latency for delivering alarm signals from sensors, or for playing audio announcements should be smaller than certain predefined thresholds. Or, the accuracy of clock synchronization between different devices, should be guaranteed under the given physical implementation of the system.

The HCS case study poses challenges that require component-based design techniques, since it involves heterogeneous components and communication mechanisms, e.g. streaming based on the data-flow paradigm as well as event driven computation and interaction. Its modeling needs combination of executable and analytic models especially for performance evaluation and analysis of non-functional properties.

We have modeled an instance of the HCS system in BIP. As shown in Figure 1, the system consists of one *Server* connected to a daisy chain of four NACs, addressed $0 \cdots 3$, and several devices. Devices are connected in daisy chains with the NACs, the length of each chain being limited to four in our example. For simplicity, devices are addressed (i, j), where i is the address of the NAC and j is the address of the device. The model contains three types of devices, namely *Audio Player*, *Video Camera* and *Smoke Sensor*. The devices connected to NAC(0) and NAC(2) have similar topology. The first two daisy-chains consist of only *Audio Player* devices. The third daisy-chain ends with a *Smoke Sensor*, and the fourth daisy-chain consists of just one *Video Camera*. The devices connected to NAC(1) and NAC(3) have exactly the same topology, consisting of several *Audio Player* and one *Smoke Sensor* devices.

The system depicted in Figure 1 contains 58 devices in total. The BIP model contains 297 atomic components, 245 clocks, and its state-space is of order 2^{3000}.

The size of the BIP code for describing the system is 2468 lines, which is translated to 9018 lines in C. A description of the key components is given in [3].

5 Experiments on the HCS

One of the core applications of the HCS case study is the PTP protocol, which allows the synchronization of the clocks of the various devices with the one of the server. It is important that this synchronization occurs properly, i.e., the difference between the clock of the server and the one of any device is bounded by a small constant. Studying this problem is the subject of this section. Since the BIP model for the HCS is extremely large (number of components, size of the state space, ...), there is no hope to analyse it with an exhaustive verification technique. Here, we propose to apply our stochastic abstraction. Given a specific device, we will proceed in two steps. First, we will conduct simulations on the entire system in order to learn the probability distribution on the communication delays between this device and the server. Second, we will use this information to build a stochastic abstraction of the application on which we will apply statistical model checking. We start with the stochastic abstraction for the PTP.

5.1 The Precision Time Protocol IEEE 1588

The Precision Time Protocol [1] has been defined to synchronize clocks of several computers interconnected over a network. The protocol relies on multicast communication to distribute a reference time from an accurate clock (*the master*) to all other clocks in the network (*the slaves*) combined with individual offset correction, for each slave, according to its specific round-trip communication delay to the master. The accuracy of synchronization is negatively impacted by the jitter (i.e., the variation) and the asymmetry of the communication delay between the master and the slaves. Obviously, these delay characteristics are highly dependent on the network architecture as well as on the ongoing network traffic.

We present below the abstract stochastic model of the PTP protocol between a device and the server in the HCS case study. The model consists of two (deterministic) application components respectively, the master and the slave clocks, and two probabilistic components, the media, which are abstraction of the communication network between the master and the slave. The former represent the behaviour of the protocol and are described by extended timed i/o-automata. The latter represent a random transport delay and are simply described by probabilistic distributions. Recap that randomization is used to represent the context, i.e., behaviors of other devices and influence of these behaviors on those of the master and the device under consideration.

The time of the master process is represented by the clock variable θ_m. This is considered the reference time and is used to synchronize the time of the slave clock, represented by the clock variable θ_s. The synchronization works as follows. Periodically, the master broadcast a *sync* message and immediately after a *followUp* message containing the time t_1 at which the *sync* message has been sent.

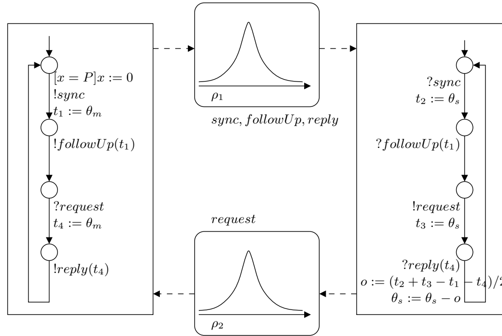


Fig. 2. Abstract stochastic PTP between the server and a device

Time t_1 is observed on the master clock θ_m. The slave records in t_2 the reception time of the *sync* message. Then, after the reception of the *followUp*, it sends a delay *request* message to the master and records its emission time t_3. Both t_2 and t_3 are observed on the slave clock θ_s. The master records on t_4 the reception time of the *request* message and sends it back to the slave on the *reply* message. Again, t_4 is observed on the master clock θ_m. Finally, upon reception of *reply*, the slave computes the offset between its time and the master time based on $(t_i)_{i=1,4}$ and updates its clock accordingly. In our model, the offset is computed differently in two different situations. In the first situation, which is depicted in Figure 2, the average delays from master to slave and back are supposed to be equal i.e., $\mu(\rho_1) = \mu(\rho_2)$. In the second situation, delays are supposed to be asymmetric, i.e., $\mu(\rho_1) \neq \mu(\rho_2)$. In this case, synchronization is improved by using an extra offset correction which compensate for the difference, more precisely, $o := (t_2 + t_3 - t_1 - t_4)/2 + (\mu(\rho_2) - \mu(\rho_1))/2$. This offset computation is an extension of the PTP specification and has been considered since it ensures better precision when delays are not symmetric (see Section 5).

Encoding the abstract model of timed i/o-automata given in Figure 2 in BIP is quite straightforward and can be done with the method presented in [2]. The distribution on the delay is implemented as a new C function in the BIP model. It is worth mentioning that, since the two i/o automata are deterministic, the full system depicted in Figure 2 is purely stochastic.

The accuracy of the synchronization is defined by the absolute value of the difference between the master and slave clocks $|\theta_m - \theta_s|$, during the time. Our aim is to check the (safety) property of bounded accuracy ϕ_Δ, that is, *always* $|\theta_m - \theta_s| \leq \Delta$ for arbitrary fixed non-negative real Δ.

Finally, a simpler version of this protocol has been considered (see [3]) and analyzed. In that study, delay components have been modeled using non-deterministic timed i/o automata as well and represent arbitrary delays bounded in some intervals $[L, U]$. It has been shown that, if the clock drift is negligible, the best accuracy $\Delta^\star$ that can be obtained using PTP is respectively $\frac{U-L}{2}$ in the symmetric case, and $\frac{U_1+U_2-L_1-L_2}{4}$ in the asymmetric case. That is, the property of bounded accuracy holds trivially iff $\Delta \geq \Delta^*$.

5.2 Model Simulations

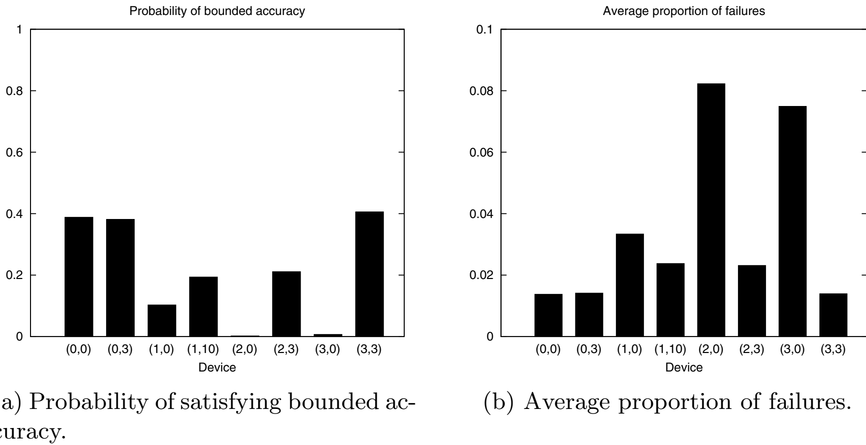
In this section, we describe our approach to learn the probability distribution over the delays. Consider the server and a given device. In a first step, we run simulations on the system and measure the end-to-end delays of all PTP messages between the selected device and the server. For example, consider the case of delay *request* messages and assume that we made 33 measures. The result will be a series of delay values and, for each value, the number of times it has been observed. As an example, delay 5 has been observed 3 times, delay 19 has been observed 30 times. The probability distribution is represented with a table of 33 cells. In our case, 3 cells of the table will contains the value 5 and 30 will contain the value 19. The BIP engine will select a value in the table following a uniform probability distribution.

According to our experiments, 2000 delay measurements are enough to obtain an accurate estimation of the probability distribution. However, for confidence reasons, we have conducted 4000 measurements. We have also observed that the value of the distribution clearly depends on the position of the device in the topology (see [3] for an illustration). It is worth mentioning that running one single simulation allowing 4000 measurements of the delay of PTP frames requires running the PTP protocol with an increased frequency i.e., the default PTP period (2 minutes) being far too big compared with the period for sending audio/video packets (tens of milliseconds). Therefore, we run simulations where PTP is executed once every 2 milliseconds and, we obtain 4000 measurements by simulating approximately 8 seconds of the global system lifetime. Each simulation uses microsecond time granularity and takes around 40 minutes on a Pentium 4 running under a Linux distribution.

5.3 Experiments on Precision Estimation for PTP

Three sets of experiments are conducted. The first one is concerned with the bounded accuracy property (see Section 5.1). In the second one, we study average failure per execution for a given bound. Finally, we study the influence of drift on the results.

Property 1: Synchronization. Our objective is to compute the smallest bound Δ under which synchronization occurs properly for any device. We start with an experiment that shows that the value of the bound depends on the place of the device in the topology. For doing so, we use $\Delta = 50\mu s$ as a bound and then compute the probability for synchronization to occur properly for all the devices. In the paper, for the sake of presentation, we will only report on a sampled set of devices: $(0,0)$, $(0,3)$, $(1,0)$, $(1,10)$, $(2,0)$, $(2,3)$, $(3,0)$, $(3,3)$, but our global observations extend to any device. We use PESTIMATION with a confidence of 0.1. The results, which are reported in Figure 3a, show that the place in the topology plays a crucial role. Device $(3,3)$ has the best probability value and Device $(2,0)$ has the worst one. All the results in Figure 3a have been conducted on the model with asymmetric delays. For the symmetric case, the probability



(a) Probability of satisfying bounded accuracy.

(b) Average proportion of failures.

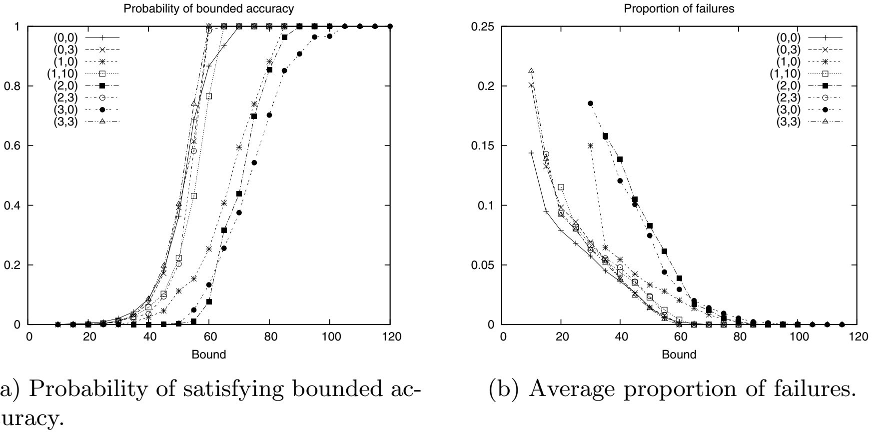
Fig. 3. Probability of satisfying the bounded accuracy property and average proportion of failures for a bound $\Delta = 50\mu s$ and the asymmetric version of PTP

values are much smaller. As an example, for Device $(0, 0)$, it decreases from 0.388 to 0.085. The above results have been obtained in less than 4 seconds. As a second experiment, we have used SPRT and SSP to validate the probability value found by PESTIMATION with a higher degree of confidence. The results, which are presented in Table 1 for Device $(0, 0)$, show that SPRT is faster than SSP and PESTIMATION.

Table 1. Number of simulations / Amount of time required for PESTIMATION, SSP and SPRT

Precision	10^{-1}		10^{-2}		10^{-3}	
Confidence	10^{-5}	10^{-10}	10^{-5}	10^{-10}	10^{-5}	10^{-10}
PESTIMATION	4883 $17s$	9488 $34s$	488243 $29m$	948760 $56m$	48824291 $> 3h$	94875993 $> 3h$
SSP	1604 $10s$	3579 $22s$	161986 $13m$	368633 $36m$	16949867 $> 3h$	32792577 $> 3h$
SPRT	316 $2s$	1176 $7s$	12211 $53s$	22870 $1m38s$	148264 $11m$	311368 $31m$

Our second step was to estimate the best bound. For doing so, for each device we have repeated the previous experiments for values of Δ between $10\mu s$ and $120\mu s$. Figure 4a gives the results of the probability of satisfying the bounded accuracy property as a function of the bound Δ for the asymmetric version of PTP. The figure shows that the smallest bound which ensure synchronization for any device is $105\mu s$ (for Device $(3, 0)$). However, devices $(0, 3)$ and $(3, 3)$ already satisfy the property with probability 1 for $\Delta = 60\mu s$. A comparison between SSP and PESTIMATION is given in [3].



(a) Probability of satisfying bounded accuracy.

(b) Average proportion of failures.

Fig. 4. Probability of satisfying the bounded accuracy property and average proportion of failures as functions of the bound Δ for the asymmetric version of PTP

The above experiments have been conducted assuming simulations of 1000 BIP interactions and 66 rounds of the PTP protocol. Since each round of the PTP takes two minutes, this also corresponds to 132 minutes of the system's life time. We now check whether the results remain the sames if we lengthen the simulations and hence system's life time. Figure 5 shows, for Devices $(0, 0)$ and $(3, 0)$, the probability of synchronization for various values of Δ and various length of simulations (1000, 4000, 8000 and 10000 (660 minutes of system's life time) steps). We used PESTIMATION with a precision and a confidence of 0.1. The best bounds do not change. However, the longer the simulations are, the more the probability tends to be either 0 or 1 depending on the bound.

Property 2: Average failure. In the previous experiment, we have computed the best bound to guarantee the bounded accuracy property. It might be the case that the bound is too high regarding the user's requirements. In such case, using the above results, we can already report on the probability for synchronization to occur properly for smaller values of the bound. We now give a finer answer by quantifying the average and worst number of failures in synchronization that occur *per simulation* when working with smaller bounds. For a given simulation, the *proportion of failures* is obtained by dividing the number of failures by the number of rounds of PTP. We will now estimate, for a simulation of 1000 steps (66 rounds of the PTP), the average and worst value for this proportion. To this purpose, we have measured (for each device) this proportion on 1199 simulations with a synchronization bound of $\Delta = 50\mu s$. As an example, we obtain average proportions of 0.036 and 0.014 for Device $(0, 0)$ using the symmetric and asymmetric versions of PTP respectively. As a comparison, we obtain average proportions of 0.964 and 0.075 for Device $(3, 0)$. The average proportion of failures with the bound $\Delta = 50\mu s$ and the asymmetric version of PTP is given in Figure 3b. Figure 6a presents, for the sampled devices, the worst proportion of

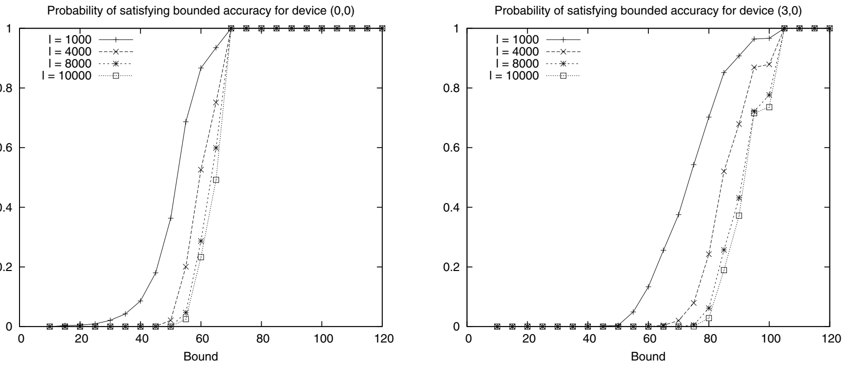
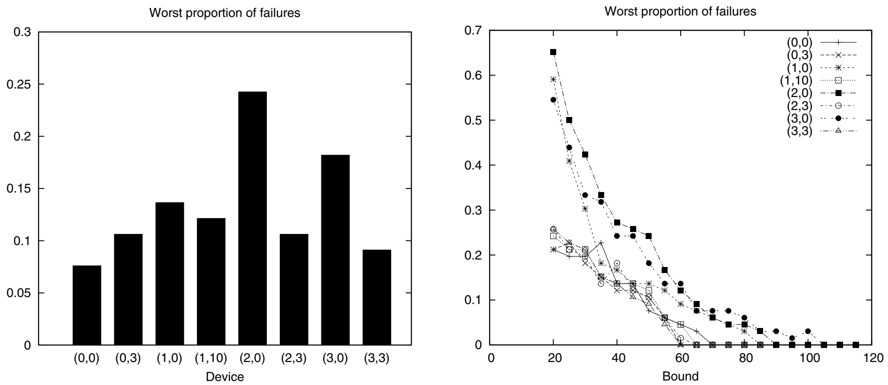


Fig. 5. Evolution of the probability of satisfying the bounded accuracy property with the length of the simulations for the asymmetric version of PTP



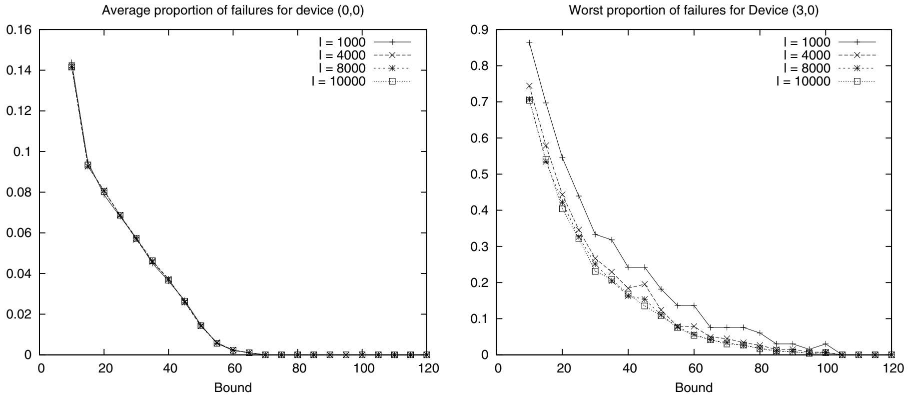
(a) Worst proportion of failures for bound $\Delta = 50\mu s$.

(b) Worst proportion of failures as a function of the bound Δ.

Fig. 6. Worst proportion of failures for the industrial bound $\Delta = 50\mu s$ and as a function of the bound Δ for the asymmetric version of PTP

failures using the asymmetric version of PTP. The worst value is 0.25, which is obtained for Device $(2, 0)$. On the other hand, the worst value is only 0.076 for Device $(0, 0)$. The experiment, which takes about 6 seconds per device, was then generalized to other values of the bound. Figures 4b and 6b give the average and worst proportion of failure as a function of the bound.

The above experiment gives, for several value of Δ and each device, the worst failure proportion with respect to 1199 simulations. We have also used PESTIMATION with confidence of 0.1 and precision of 0.1 to verify that this value remains the same whatever the number of simulations is. The result was then validated using SSP with precision of 10^{-3} and confidence of 10^{-10}. Each experiment took approximately two minutes. Finally, we have conducted experiments to check whether the results remain for longer simulations. Figure 7a shows that



(a) Average proportion of failure for Device (0,0).

(b) Worst proportion of failure for Device (3,0).

Fig. 7. Evolution of the average and worst proportion of failures with the length of the simulations for the asymmetric version of PTP

the average proportion does not change and Figure 7b shows that the worst proportion decreases when the length of the simulation increases.

Clock Drift. We have considered a modified version of the stochastic PTP model with drifting clocks. Drift is used to model the fact that, due to the influence of the hardware, clocks of the master and the device may not progress as the same rate. In our model, drift is incorporated as follows: each time the clock of the server is increased by 1 time unit, the clock of the device is increased by $1 + \varepsilon$ time units, with $\varepsilon \in [-10^{-3}, 10^{-3}]$. Using this modified model, we have re-done the experiments of the previous sections and observed that the result remains almost the same. This is not surprising as the value of the drift is significantly smaller than the communication jitter, and therefore it has less influence on the synchronization. A drift of 1 time unit has a much higher impact on the probability. As an example, for Device $(0, 0)$, it goes from a probability of 0.387 to a probability of 0.007. It is worth mentioning that exhaustive verification of a model with drifting clocks is not an easy task as it requires to deal with complex differential equations. When reasoning on one execution at a time, this problem is avoided.

References

1. IEEE - IEC 61588: Precision clock synchronization protocol for networked measurement and control systems (2004)
2. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Systems in BIP. In: SEFM 2006, Pune, India, September 2006, pp. 3–12 (2006)

3. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. Research Report RR-7238, INRIA (March 2010), http://hal.inria.fr/inria-00466158/PDF/RR-7238.pdf
4. The BIP Toolset, http://www-verimag.imag.fr/~async/bip.php
5. Clarke, E.M., Donzé, A., Legay, A.: Statistical model checking of mixed-analog circuits with an application to a third order delta-sigma modulator. In: HVC 2008. LNCS, vol. 5394, pp. 149–163. Springer, Heidelberg (2008) (to appear)
6. Clarke, E.M., Faeder, J.R., Langmead, C.J., Harris, L.A., Jha, S.K., Legay, A.: Statistical model checking in biolab: Applications to the automated analysis of t-cell receptor signaling pathway. In: Heiner, M., Uhrmacher, A.M. (eds.) CMSB 2008. LNCS (LNBI), vol. 5307, pp. 231–250. Springer, Heidelberg (2008)
7. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
8. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004)
9. Hoeffding, W.: Probability inequalities. Journal of the American Statistical Association 58, 13–30 (1963)
10. Jansen, D.N., Katoen, J.P., Oldenkamp, M., Stoelinga, M., Zapreev, I.S.: How fast and fat is your probabilistic model checker? an experimental performance comparison. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 69–85. Springer, Heidelberg (2008)
11. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A bayesian approach to model checking biological systems. In: Degano, P., Gorrieri, R. (eds.) CMBS 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009)
12. Laplante, S., Lassaigne, R., Magniez, F., Peyronnet, S., de Rougemont, M.: Probabilistic abstraction for model checking: An approach based on property testing. ACM Trans. Comput. Log. 8(4) (2007)
13. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
14. Wald, A.: Sequential tests of statistical hypotheses. Annals of Mathematical Statistics 16(2), 117–186 (1945)
15. Younes, H.L.S.: Verification and Planning for Stochastic Processes with Asynchronous Events. Ph.D. thesis, Carnegie Mellon (2005)
16. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. STTT 8(3), 216–228 (2006)

Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude

Peter Csaba Ölveczky[1], Artur Boronat[2], and José Meseguer[3]

[1] University of Oslo
[2] University of Leicester
[3] University of Illinois at Urbana-Champaign

Abstract. AADL is a standard for modeling embedded systems that is widely used in avionics and other safety-critical applications. However, AADL lacks a formal semantics, and this severely limits both unambiguous communication among model developers, and the development of simulators and formal analysis tools. In this work we present a formal object-based real-time concurrent semantics for a behavioral subset of AADL in rewriting logic, which includes the essential aspects of its behavior annex. Our semantics is directly executable in Real-Time Maude and provides an AADL simulator and LTL model checking tool called *AADL2Maude*. *AADL2Maude* is integrated with OSATE, so that OSATE's code generation facility is used to automatically transform AADL models into their corresponding Real-Time Maude specifications. Such transformed models can then be executed and model checked by Real-Time Maude. We present our semantics, and two case studies in which safety-critical properties are analyzed in *AADL2Maude*.

1 Introduction

AADL [15] is both a modeling language for real-time embedded systems and an international standard widely used in industry. It has features to model the real-time aspects of embedded systems and to represent both the software and hardware architectures of the components making up such systems. It does however lack a formal semantics. This lack is particularly important for real-time embedded systems, because many of them —in areas such as avionics, motor vehicles, and medical systems— are *safety-critical* systems, whose failures may cause great damage to persons and/or valuable assets. Furthermore, AADL models are not executable, which limits not just the possibility of formal analysis of their safety and liveness properties, but even the possibility of simulating them.

It seems clear that overcoming these limitations of AADL is highly desirable, but requires in an essential way the use of formal methods, because in the absence of a precise mathematical semantics any pretense of achieving formal verification is meaningless. Furthermore, these formal methods should be supported by tools that are integrated into the AADL tool chain. A further, highly desirable goal is to have a formal semantics of AADL that can be used to automatically generate *formal executable specifications* of AADL models, since the first and most

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 47–62, 2010.
© IFIP International Federation for Information Processing 2010

basic way of analyzing AADL models should be the capacity to *simulate* such models; and since such formal executable specifications can then also be used for automatic verification of safety and liveness properties by *model checking*.

This paper reports on our experience in defining an object-based real-time concurrent formal semantics for a substantial subset of AADL in the Real-Time Maude formal specification language [14]; and in directly using this semantics to simulate and formally analyze AADL models. We have found Real-Time Maude particularly well suited for this task for the following reasons:

- *Support for nested objects.* AADL models are structured in nested hierarchies of components. Much would be lost in translation if such structure were not preserved. Real-Time Maude's support for object classes with a "Russian dolls" nested structure (see [10]), provides an essentially isomorphic formal counterpart for an AADL model.
- *Support for real-time concurrency.* All real-time aspects of AADL, as well as the concurrent interactions between AADL components, can be directly and naturally formally modeled by means of *real-time rewrite theories*.
- *Wide range of formal analysis capabilities.* By automating the AADL formal semantics with the *AADL2Maude* tool, one can automatically generate formal executable specifications of AADL models in Real-Time Maude for *simulation*, *reachability analysis*, and *LTL model checking purposes*.
- *Completeness of the formal analysis.* In spite of the generality of the AADL models, their object-based semantics ensures that *time-bounded LTL properties* are *decidable* under very mild checkable conditions [13].

Our Contribution. To the best of our knowledge (see the related work discussion in Section 4), our work is the first that provides a *formal executable semantics* for AADL models with different modes, and whose thread behavior is specified in AADL's behavior annex; it is also the first that supports simulation, reachability, and LTL model checking of such models *directly in the semantic formalism itself* through the *AADL2Maude* tool. *AADL2Maude* is an OSATE plug-in that uses OSATE's code generation facility to automatically generate Real-Time Maude specifications from AADL models. Furthermore, our semantics directly supports *hierarchical objects* that communicate asynchronously with each other in real time and capture the hierarchical nature of AADL components. This makes the representational distance between the original AADL model and its rewriting logic semantics quite small, making it easier to understand the results of formal analysis. User-friendliness is also enhanced by a syntax for state predicates based on AADL notation to ease the specification of LTL properties. Finally, we summarize two case studies, one on safe interoperation of medical devices and one on the safety of avionics systems, demonstrating the usefulness of this semantics and tool in concrete examples.

The paper is organized as follows. Section 2 gives a brief introduction to AADL and Real-Time Maude. Section 3 presents the Real-Time Maude semantics of a behavioral subset of AADL, and shows how AADL models can be formally analyzed in *AADL2Maude*. Section 4 discusses related work on the use of formal methods for AADL, and Section 5 presents some concluding remarks.

2 Preliminaries on AADL and Real-Time Maude

AADL. The *Architecture Analysis & Design Language* (AADL) [15] is an industrial standard used in avionics, aerospace, automotive, medical devices, and robotics communities to describe a performance-critical embedded real-time system as an assembly of software components mapped onto an execution platform.

An AADL model describes a system as a hierarchy of hardware and software components. A component is defined by its *name*, its *interface* consisting of input and output ports, its *subcomponents* and their interaction, and other type-specific *properties*. System components are the top-level components, and can consist of other system components as well as of hardware and software components. Hardware components include: *processor* components that schedule and execute threads; *memory* components; *device* components representing devices like sensors and actuators that interface with the environment; and *bus* components that interconnect processors, memory, and devices. Software components include: *thread* components modeling the application software to be executed; *process* components defining protected memory that can be accessed by its thread subcomponents; and *data* components representing data types. In AADL, thread behavior is typically described using AADL's *behavior annex* [6], which models programs as transition systems with local state variables.

An AADL model specifies how the different components interact and are integrated to form a complete system. The AADL standard also describes the runtime mechanisms for handling message and event passing, synchronized access to shared resources, thread scheduling when several threads run on the same processor, and dynamic reconfiguration that are specified by *mode transitions*.

AADL has a MOF meta-model, and the OSATE modeling environment provides a set of plug-ins for front-end processing of AADL models on top of Eclipse.

Real-Time Maude. A Real-Time Maude [14] *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [5] theory with Σ a signature[1] and E a set of *confluent and terminating conditional equations*. (Σ, E) specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.
- IR is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, written with syntax `rl [`l`] : `t` => `t', where l is a *label*. Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t'. The rules are applied *modulo* the equations E.[2]
- TR is a set of *tick (rewrite) rules*, written with syntax

  `rl [`l`] : {`t`} => {`t'`} in time `τ`.`

[1] I.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*.

[2] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

that model time elapse. `{_}` encloses the global state, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The Real-Time Maude syntax is fairly intuitive. For example, a function symbol f is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Equations are written with syntax `eq` t `=` t', and `ceq` t `=` t' `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars`. We refer to [5] for more details on the syntax of Real-Time Maude.

In *object-oriented* Real-Time Maude modules, a *class* declaration

`class` C `|` att_1 `:` s_1`,` ... `,` att_n `:` s_n`.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a state is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a *concurrent* object-oriented system, the state is a term of sort `Configuration`. It has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] :  < O : C | a1 : 0, a2 : y, a3 : w, a4 : z >  =>
          < O : C | a1 : T, a2 : y, a3 : y + w, a4 : z >
```

defines a parametrized family of transitions which can be applied whenever the attribute `a1` of an object `O` of class `C` has the value `0`, with the effect of altering the attributes `a1` and `a3` of the object. "Irrelevant" attributes (such as `a4`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule (or equation).

A *subclass* inherits all the attributes and rules of its superclasses.

Formal Analysis. A Real-Time Maude specification is *executable*, and the tool offers a variety of formal analysis methods. The *rewrite* command simulates *one* fair behavior of the system *up to a certain duration.* The *search* command uses a breadth-first strategy to analyze all possible behaviors of the system, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state. The command which searches for n such states has syntax `(utsearch [`n`]` t `=>*` *pattern* `such that` *cond* `.)`.

Real-Time Maude also extends Maude's *linear temporal logic model checker* to check whether each behavior, possibly up to a certain time bound, satisfies a temporal logic formula. *State propositions*, possibly parametrized, can be predicates characterizing properties of the state and/or properties of the global time of the system. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). A time-bounded model

checking command for initial state t and temporal logic formula *formula* has syntax (mc t |=t *formula* in time <= τ .).

3 Real-Time Maude Semantics for a Subset of AADL

This section gives an overview of the Real-Time Maude semantics for a behavioral subset of AADL. Section 3.1 presents the chosen subset of AADL, Section 3.2 presents its Real-Time Maude semantics, Section 3.3 explains how AADL models can be simulated and formally analyzed in Real-Time Maude and illustrates such formal analysis with a medical devices example, and Section 3.4 summarizes an avionics safety example. A technical report giving more details, the entire executable Real-Time Maude semantics, and some AADL models and the corresponding automatically generated Real-Time Maude models are all available at http://www.ifi.uio.no/RealTimeMaude/AADL.

3.1 Overview of a Behavioral Subset of AADL

In AADL, a system is modeled as a collection of software and hardware components. Since we focus on the software parts of AADL, the following description only deals with the software components and features.

A component is given by its *type* and its *implementation*. A component type specifies the component's *interface* in terms of *features* and *properties*. In the software portion, features are just input and output *ports*. A component implementation specifies the internal structure of the component in terms of a set of *subcomponents*, a set of *connections* linking the ports of the subcomponents, and *modes* that represent operational states of components. *System* components are the top level components. A *process* component contains a set of *thread* components that define the dynamic behavior of the process.

Connections link ports to enable the exchange of data and events among components. A port is either a *data* port, an *event* port, or an *event data* port. *Buffers* associated to event ports and event data ports support queuing of, respectively, "events" and message data, while buffers of *data* ports only keep the latest data.

Modes represent the operational states of components. A component can have mode-specific property values, subcomponents, and connections. Mode transitions are triggered by events.

The *dispatch protocol* property of a thread determines when the thread is executed. A *periodic* thread is activated at time intervals of the specified period T; an *aperiodic* thread is activated when an event arrives at a port of the thread; a *sporadic* thread is activated when an event arrives *and* the interval between two dispatches is at least T; and a *background* thread is always active.

The dynamic behavior of a thread is defined using AADL's *behavior annex* [6]. Given finite sets of *states* and *state variables*, the behavior of a thread is defined by a set of state transitions of the form s -[*guard*]-> s' {*actions*}, where s and s' are states, and where *guard* is a Boolean condition on the values of the

state variables and/or the presence of events or data in the thread's input ports. The *actions* that are performed when a transition is applied may update the state variables, generate new outputs, and/or suspend the thread for a given amount of time. Actions are built from basic actions using a small set of control structures allowing sequencing, conditionals, and finite loops. When a thread is activated, an enabled transition is nondeterministically selected and applied; if the resulting state s' is not a *complete* state, another transition is applied, and so on, until a complete state is reached (or the thread is suspended).

An AADL Example. As an example of a specification within our subset of AADL, consider a network of medical devices, consisting of a *controller*, a *ventilator machine* that assists a patient's breathing during surgery, and an *X-ray* device. Whenever a button is pushed to take an X-ray, and the ventilator machine has *not* paused in the past 10 minutes, the ventilator machine should pause for two seconds, starting one second after the button is pushed, and the X-ray should be taken after two seconds. To execute the system, we add a *test activator* that pushes the button every second.

The following AADL model was developed by Min-Young Nam at UIUC.

The entire system `Wholesys` is a closed system that does not have any features (i.e., ports) to the outside world. Hence, its *type* (interface) is empty:

```
system Wholesys
end Wholesys;
```

The *implementation* of the entire system describes the architecture of the system, with four subcomponents and the connections linking these subcomponents:

```
system implementation Wholesys.imp
 subcomponents TestActivator: system TA.impl;   Xray: system XM.impl;
               Controller: system CTRL.impl;    Ventilator: system VM.impl;
 connections
  C01: event data port Controller.xmContrOutput -> Xray.ctrlInput;
  C02: event data port Controller.vmContrOutput -> Ventilator.ctrlInput;
  C03: event data port Ventilator.feedback -> Controller.feedback;
  C04: event data port TestActivator.pressEvent -> Controller.commandInput;
end Wholesys.imp;
```

The test activator, which generates an event every second, is an instance of a *system* of type `TA`, having as interface the output port `pressEvent`. Its implementation consists of a process `taPr`, which again consists of a single thread `taTh` that is an instance of the following `taThread.impl`:

```
thread taThread
  features      pressEvent: out event data port Behavior::integer;
  properties    Dispatch_Protocol => periodic;    Period => 1 sec;
end taThread;
```

```
thread implementation taThread.impl
  annex behavior_specification {**
    states          s0: initial complete state;
    transitions     s0 -[ ]-> s0 {pressEvent!(1);}; **};
end taThread.impl;
```

The thread `taTh` is dispatched every second. When the thread is dispatched, the transition is applied once (since the resulting state `s0` is a complete state), and the action performed is to output the value `1` through the port `pressEvent`.

3.2 Real-Time Maude Semantics of AADL

This section outlines the object-based real-time rewriting logic semantics for the behavioral subset of AADL presented in Section 3.1. We first show how an AADL model is represented in Real-Time Maude, and then formalize the real-time concurrent semantics of AADL models.

Representing AADL Models in Real-Time Maude. The semantics of a component-based language can naturally be defined in an object-oriented style, where each component instance is modeled as an object. The hierarchical structure of AADL components is reflected in the nested structure of objects, in which an attribute of an object contains its subcomponents as a multiset of objects.

Any AADL component instance is represented as an object instance of a subclass of the following class `Component`, which contains the attributes common to all kinds of components (systems, processes, threads, etc.):

```
class Component | features : Configuration,   subcomponents : Configuration,
                  properties : Properties,     connections : ConnectionSet,
                  modes : Modes,               inModes : ModeNameSet .
```

The attribute `features` denotes the features of a component (i.e., its ports), represented as a multiset of `Port` objects (see below); `subcomponents` denotes the subcomponents of the object; `properties` denotes its *properties*, such as the dispatch protocol for threads; `connections` denotes the set of port connections of the object (see below); `modes` contains the object's mode transition system; and `inModes` gives the set of modes (of the immediate supercomponent) in which the component is available (if the component is not a mode-specific subcomponent of the containing component, then this attribute has the value `allModes`).

In our AADL subset, the classes `System` and `Process`, denoting system and process components, do not have other attributes than those they inherit from their `Component` superclass. The `Thread` class is declared as follows:

```
class Thread | behavior : ThreadBehavior,    status : ThreadStatus,
               deactivated : Bool .
subclass Thread < Component .
```

The `behavior` attribute denotes the transition system associated with the thread. The `status` indicates the current status of the thread (`active`, `completed`, `suspended`, etc.). The attribute `deactivated` indicates whether the thread is deactivated because it is not in the current "active" modes of the system.

Ports and connections. A port is modeled as an object instance of a subclass of the class `Port`, whose subclasses define outgoing and incoming ports, as well as data, event, and event data ports. See [12] for details. An immediate level-up connection, linking an outgoing port P in a subcomponent C to the outgoing port P' in the "current" component, is modeled as a term C `.` P `-->` P'. Immediate same-level and level-down connections are terms of the forms, respectively, P_1 `-->` P_2 and P `-->` C `.` P'.

Representing Thread Behavior. The transition system associated with a thread is modeled as a term of the form:

```
states            current: s complete: s_1 ... s_k other: s_{k+1} ... s_n
state variables   var_1 |-> val_1    ...    var_m |-> value_m
transitions       s -[guard]-> s' {actions} ; ... ; s'' -[guard']-> s' {actions'}
```

We have also defined some additional "syntactic sugar" functions (e.g., allowing the definition of `initial` states, omitting the declaration of the store when no state variables are declared) that reduce to the above form. The sets of transitions, locations, and variable mappings have the structure of a multiset, using a multiset union operator that is declared to be associative and commutative.

Translating an AADL Model into an Object-Based Real-Time Maude Module. One main goal of our semantics is to make the "representational distance" between an AADL model and the corresponding Real-Time Maude module as small as possible. In particular, this simplifies an automatic translation from an AADL model to a similar-looking Real-Time Maude module.

Consider a type declaration of a component (`System`, `Process`, or `Thread`):

```
system typeName [features: ports]  [properties: properties]  end typeName;
```

This declaration binds $typeName$ to a set of ports and a set of properties. We can therefore consider `system` as a function that, given a name, returns the interface of that name; hence the above AADL declaration translates to the equation

```
eq system(typeName) = features ports_RTM properties properties_RTM .
```

where $ports_{RTM}$ denotes the Real-Time Maude representation of *ports*.

A component *implementation*, such as

```
system implementation typeName.implName
   ...
end typeName.implName;
```

defines an component *template*, which is instantiated to a concrete instance of the component in AADL declarations of the form (the 'in modes' part is optional)

```
instanceName: system typeName.implName [in modes (mode names)]
```

Therefore, the above implementation declaration translates to an equation

```
var INSTANCE-NAME : Oid .   var MNS : ModeNameSet .
eq INSTANCE-NAME system typeName . implName in modes MNS =
   < INSTANCE-NAME : System | features : features(system(typeName)),
                              properties : properties(system(typeName)),
                              inModes : MNS,  subcomponents : ...,   ... >
```

In addition, the "generic" equation

```
var SI : SystemId .   var SN : SystemName .   var IN : ImplName .
eq SI system SN . IN = SI system SN . IN in modes allModes .
```

allows us to declare component instances that are not mode-dependent. The above AADL component instance declaration therefore translates to the term

instanceName `system` *typeName* . *implName*.

Example 1. Consider the AADL model of the medical system in Section 3.1. The definition of the implementation `Wholesys.imp` in Section 3.1 is translated to

```
eq INSTANCE-NAME system Wholesys . imp in modes MNS =
   < INSTANCE-NAME : System |
      modes : noModes,   inModes : MNS,
      features : features(system(Wholesys)),
      properties : properties(system(Wholesys)),
      subcomponents :
        (TestActivator system TA . impl)    (Xray system XM . impl)
        (Controller system Controller . impl)  (Ventilator system VM . impl),
      connections :
        (Controller . xmContrOutput --> Xray . ctrlInput) ;
        (Controller . vmContrOutput --> Ventilator . ctrlInput) ;
        (Ventilator . feedback --> Controller . feedback) ;
        (TestActivator . pressEvent --> Controller . commandInput) > .
```

The test activator thread `taThread` and its implementation `taThread.impl` are translated as follows:

```
eq thread(taThread) = features (pressEvent out event data thread port)
                      properties DispatchProtocol(Periodic); Period(1 Sec).

eq  INSTANCE-NAME thread taThread . impl in modes MNS =
  < INSTANCE-NAME : Thread |
       modes : noModes,   inModes : MNS,
       features : features(thread(taThread)),
       subcomponents : none,  connections : none,
       properties : properties(thread(taThread)),
       behavior :  states   initial: s0  complete: s0
                   transitions  s0 -[]-> s0 {(pressEvent ! (1))} > .
```

where `pressEvent out event data thread port` is defined to be the object `< pressEvent : OutEventDataThreadPort | buffer : nil >`.

Real-Time Concurrent Semantics. This section formalizes the operational semantics of AADL in Real-Time Maude. The real-time concurrent semantics is defined by equations and rewrite rules specifying "message" transportation, mode switches, thread dispatch, thread execution, and timed behavior. We give only a small sample of our semantic definitions, and refer to [12] for more details.

Thread Dispatch and Execution. The *execution status* of a thread is either *active*, *completed*, *sleeping*, or *inactive*. When a *completed* thread is dispatched, the thread enters the *active* status to perform the computation. Upon successful completion of the computation, the thread returns to the *completed* status. Once an active thread executes a *delay* action, it enters the *sleeping* status, suspends for a period of time, and becomes active after that time period. Finally, a thread is *inactive* if it is not part of the "active" mode of the system.

The following rule models the dispatch of a *periodic* and *completed* thread when the "dispatch timer," i.e., the second parameter to `periodic-dispatch` is `0`. The thread is dispatched, that is, its `status` is set to `active`, the "timer" is reset to the length `T` of its period, and the input ports are "dispatched" as well:

```
rl [periodic-dispatch] :
   < O : Thread | properties : periodic-dispatch(T, 0)  PROPS,
                  status : completed,  features : PORTS >
  =>
   < O : Thread | properties : periodic-dispatch(T, T)  PROPS,
                  status : active ,  features : dispatchInputPorts(PORTS) > .
```

The following rewrite rule specifies the execution of an *active* thread. If the thread is in state `L1`, and there is a transition from `L1` whose guard evaluates to `true`, then the transition is executed. The resulting `status` is `sleeping(...)` if the statement list `SL` contains `delay` statements; otherwise, the thread is `completed` or `inactive` if the resulting state `L2` is a complete state, and remains `active` if `L2` is not a complete state:

```
crl [apply-transition] :
    < O : Thread | status : active ,  deactivated : B,  features : PORTS,
                   behavior :
                      states current: L1 complete: LS1 others: LS2
                      state variables VAL
                      transitions (L1 -[GUARD]-> L2 {SL})  ; TRANSITIONS >
   =>
    < O : Thread | status : (if SLEEP then sleeping(SLEEP-TIME) else
                               (if (not L2 in LS1) then active else
                                (if B then inactive else completed fi) fi) fi),
                  features : NEW-PORTS,
                  behavior :
                      states current: L2 complete: LS1 others: LS2
                      state variables NEW-VALUATION
                      transitions (L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS >
    if evalGuard(GUARD, PORTS, VAL)
```

```
   /\ transResult(NEW-PORTS, NEW-VALUATION, SLEEP-TIME) :=
        executeTransition( L1 -[GUARD]-> L2 {SL}, PORTS, VAL)
   /\ SLEEP := SLEEP-TIME > 0 .
```

The function `executeTransition` executes a given transition in a state with a given set `PORTS` of ports and assignment `VAL` of the state variables. The function returns a triple `transResult(`p, σ, t`)`, where p is the state of the ports after the execution, σ denotes the resulting values of the state variables, and t is the sum of the `delays` in the transition actions. The transitions are modeled as a multiset of single transitions; therefore, *any* enabled transition can be applied in the rule.

Time Behavior. We model time elapse in the system by a single tick rule

```
crl {SYSTEM}  =>  {delta(SYSTEM, T)} in time T if T <= mte(SYSTEM) .
```

The function `delta` defines the effect of time elapse in a system, and the function `mte` defines the *m*aximal *t*ime *e*lapse possible until an action must be taken. These functions distribute over the elements in a (sub)configuration, propagate to the subcomponents of `system` and `process` components, and must be defined for single thread objects to define the time behavior of a system.

The following must be taken into account when defining these functions: (i) periodic threads must dispatch at the correct times; (ii) threads in `sleep` status must wake up when their sleep time expires; (iii) time must not elapse when there are "untreated" messages in the system, since an aperiodic thread is dispatched when it receives an event; and (iv) time cannot advance when a thread is in `active` state, as the thread should execute a transition when it is `active`.

The function `delta` modeling the effect of time elapse decreases the "timer" t in a `periodic-dispatch(T,`t`)` property of a thread, and the timer t' in the `sleeping(`t'`)` status of a thread, according to the elapsed time:

```
eq delta(< THR : Thread | subcomponents : C, status : TS, properties : PROPS >, T)
  = < THR : Thread | subcomponents : delta(C, T), status : delta(TS, T),
                     properties : delta(PROPS, T) > .

op delta : ThreadStatus Time -> ThreadStatus .
eq delta(sleeping(T), T') = sleeping(T - T').  eq delta(TS, T') = TS [owise] .
op delta : Properties Time -> Properties .
eq delta(periodic-dispatch(T,T') PROPS, T'') =
      periodic-dispatch(T, T' - T'') PROPS .
eq delta(PROPS, T) = PROPS [owise] .
```

The function `mte` (maximum time elapse) ensures that `mte` is 0 when an "untreated" message list, that is, one of the form `transfer(`ml`)`, is present in some port buffer; in addition, it ensures that time cannot advance beyond the wake-up time of a sleeping thread, or beyond the dispatch time of a periodic thread. In addition, time cannot advance when a thread is `active`:

```
eq mte(< THR : Thread | features : PORTS, subcomponents : C,
                         status : TS, properties : PROPS >)
   = min(mte(PORTS), mte(C), mte(TS), mte(PROPS)) .

eq mte(< P : Port | buffer : ML :: transfer(ML') :: ML''  >) = 0 .
eq mte(< P : Port | buffer : ML >) = INF [owise] .
op mte : ThreadStatus -> TimeInf .
eq mte(active) = 0 .  eq mte(completed) = INF .  eq mte(sleeping(T)) = T .
eq mte(inactive) = INF .
op mte : Properties -> TimeInf .
eq mte(periodic-dispatch(T, T') PROPS) = T' .  eq mte(PROPS) = INF [owise].
```

3.3 Formal Analysis of AADL Models: A Medical Devices Example

The Real-Time Maude verification model synthesized from an AADL design model can be formally analyzed in different ways. This section presents some functions allowing the user to define system properties in terms of an AADL model without having to understand its Real-Time Maude representation. We illustrate the formal analysis features with the plug-and-play interoperation of medical devices example.

Defining Initial States and Simulation. An AADL system definition declares a component template. An initial state is an instance of such a template. In the medical example, if `MAIN` is a system component name, the initial state is `{MAIN system Wholesys . impl}`. In addition, a function `initialize` is used to correctly initialize the `status` and `deactivated` attributes in the threads, since a thread may be inactive if a mode-specific component much higher in the containment hierarchy is not part of the "current" mode.

A first form of formal analysis consists of simulating *one* of the many possible system behaviors up to a given duration using *timed rewriting*:

```
Maude> (tfrew initialize({MAIN system Wholesys . impl}) in time < 20 .)
```

Reachability Analysis. Real-Time Maude's `tsearch` and `utsearch` commands can be used to analyze whether or not a *state pattern* can be reached from the initial state. To avoid requiring the user of *AADL2Maude* to know the Real-Time Maude representation of AADL models to define his/her state patterns, our tool defines some useful functions. The term

`value of` v `in component` *fullComponentName* `in` *globalComponent*

returns the value of the state variable v in the thread identified by the full component name *fullComponentName* in the system in state *globalComponent*. The full component name is defined as a `->`-separated path of component names, from the outermost to the innermost. Likewise, the term

`location of component` *fullComponentName* `in` *globalComponent*

gives the current location/state in the transition system in the given thread.

In our medical example, `MAIN -> Xray -> xmPr -> xmTh` denotes the full component name of the `xmTh` thread. The system must ensure that the ventilator machine is pausing when an X-ray is being taken, so that the X-ray is not blurred. The following search command analyzes this property by checking whether an *undesired* state, where the X-ray thread `xmTh` is in state `xray` while the ventilator thread `vmTh` is *not* in state `paused`, can be reached from the initial state (the unexpected result shows a concrete unsafe state that can be reached from the initial state):

```
Maude> (utsearch [1]
         initialize({MAIN system Wholesys . impl}) =>* {C:Configuration}
          such that
           ((location of component (MAIN -> Xray -> xmPr -> xmTh)
               in C:Configuration) == xray
            and (location of component (MAIN -> Ventilator -> vmPr -> vmTh)
                   in C:Configuration) =/= paused) .)

Solution 1    C:Configuration --> ...
```

LTL Model Checking. For LTL model checking purposes, our tool has pre-defined useful parametric atomic propositions, such as *full thread name* `@` *location*, which holds when the thread is in state *location*.

We can use time-bounded LTL model checking to verify that an X-ray must be taken within three seconds of the start of the system (this command returned a counter-example revealing a subtle and previously unknown design error):

```
Maude> (mc initialize({MAIN system Wholesys . impl}) |=t
          <> ((MAIN -> Xray -> xmPr -> xmTh) @ xray) in time <= 3 .)

Result ModelCheckResult :  counterexample( ... )
```

3.4 An Active Standby Avionics Example

The *AADL2Maude* tool has been used by Edgar Pek to verify an AADL model developed by Abdullah Al-Nayeem of an *active standby* specification by Steve Miller from Rockwell-Collins for deciding which of two computer systems is active in an aircraft [11]. The *active standby* system is a simplified example of a fault-tolerant avionics system. In *integrated modular avionics* (IMA), a cabinet is a chassis with a power supply, internal bus, and general purpose computing, I/O, and memory cards. Aircraft applications are implemented using the resources in the cabinets. There are always two or more cabinets that are physically separated on the aircraft so that physical damage (e.g., an explosion) doesn't take out the computer system. The active standby system considers the case of two cabinets and focuses on the logic of deciding which side is *active* in a setting where each side can fail, and where the user/pilot can toggle the active status of these sides.

All the desired system properties have been verified by unbounded LTL model checking of the synthesized Real-Time Maude verification model. We refer to [12] for more details on this case study.

4 Related Work

The applications of formal methods to analyze AADL models can be divided into: (i) those that handle AADL models *without* the behavior annex; (ii) those that add to an AADL model an *external behavior specification*; and (iii) those that handle AADL models whose behavior is specified in AADL's behavior annex.

Work in class (i) includes [16,7]; they focus on analyzing schedulability and/or behavior of an architectural subset of AADL where thread behavior is only characterized by dispatch protocol and execution time. Work in class (ii) includes [9,1,2,8]; they all assume that thread behavior is specified *outside* AADL, but differ on how this is done. [9] uses the Lustre synchronous language; [1] uses communicating timed automata; [2] uses rewrite rules; and [8] uses Ada.

Work in class (iii) includes [3,4,17] and our own work. The main difference between [3,4] and our work is that we give a *formal executable semantics* to an AADL model with a behavior annex specification of its thread behavior, associating to it a real-time rewrite theory. Instead, both [3] and [4] are based on *translations into imperative languages*, which are themselves in need of a formal semantics. Specifically, [3] maps AADL models into the Fiacre language, which contains assignments, conditionals, while loops and sequential composition constructs; and [4] maps AADL models to the BIP language, in which state transitions are defined using code written in C. The paper [17], like us, proposes a formal semantics, in their case in the Timed Abstract State Machine (TASM) formalism; however, they deal with a smaller subset (periodic threads, no modes) and do not support model checking analysis, for which they suggest using the Uppaal timed automata-based tool.

In summary, to the best of our knowledge our work is the first that provides a formal executable semantics for AADL models with modes, and whose thread behavior is specified in AADL's behavior annex; and also the first that supports simulation and LTL model checking of such models in the semantic formalism.

5 Conclusions

AADL's current lacks of a formal semantics and of executability are two severe limitations, particularly for certifiable safety-critical embedded systems. In this work we solve these two problems for a substantial subset of AADL by providing a formal object-oriented real-time rewriting semantics of it in Real-Time Maude, and by deriving from this semantics a tool, *AADL2Maude*, that connects the OSATE AADL tool with Real-Time Maude and supports simulation, reachability, and LTL model checking analyses of AADL models in this subset. Furthermore, we have illustrated the use of *AADL2Maude* with two case studies, one of safe medical device interoperation, and another on safety of an avionics system.

Our experience is quite encouraging, but much work remains. Increasingly larger AADL subsets should be given a formal rewriting logic semantics to achieve the goal of giving a formal semantics to the entire AADL standard and having simulation and formal analysis tools for AADL based on such a semantics. Also, further experimentation to extend and perfect our approach should

be carried out. We also plan to make it even easier for users to specify formal properties of AADL models in an AADL "formal property annex," so that such properties can be expressed solely in terms of the given AADL model.

Acknowledgments. We thank Abdullah Al-Nayeem, Min Young Nam, Lui Sha, and Mu Sun, for many fruitful discussions on AADL, Xiokang Qiu for his work on a previous prototype, and Edgar Pek for his work on the active standby example. Partial support from Rockwell Collins and Lockheed Martin, from the Research Council of Norway, and from NSF under Grant CNS 08-34709 is gratefully acknowledged.

References

1. Abdoul, T., Champeau, J., Dhaussy, P., Pillain, P.Y., Roger, J.C.: AADL execution semantics transformation for formal verification. In: ICECCS 2008. IEEE, Los Alamitos (2008)
2. Benammar, M., Belala, F., Latreche, F.: AADL behavioral annex based on generalized rewriting logic. In: Proc. RCIS 2008. IEEE, Los Alamitos (2008)
3. Berthomieu, B., Bodeveix, J.P., Chaudet, C., Dal-Zilio, S., Filali, M., Vernadat, F.: Formal verification of AADL specifications in the Topcased environment. In: Kordon, F., Kermarrec, Y. (eds.) Reliable Software Technologies – Ada-Europe 2009. LNCS, vol. 5570, pp. 207–221. Springer, Heidelberg (2009)
4. Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J.: Translating AADL into BIP - application to the verification of real-time systems. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 5–19. Springer, Heidelberg (2009)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex - experiments and roadmap. In: ICECCS. IEEE, Los Alamitos (2007)
7. Gui, S., Luo, L., Li, Y., Wang, L.: Formal schedulability analysis and simulation for AADL. In: ICESS 2008. IEEE, Los Alamitos (2008)
8. Hugues, J., Zalila, B., Pautet, L., Kordon, F.: From the prototype to the final embedded system using the Ocarina AADL tool suite. ACM Trans. Embedded Comput. Syst. 7(4) (2008)
9. Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Proc. EMSOFT 2007. ACM, New York (2007)
10. Meseguer, J., Talcott, C.: Semantic models for distributed object reflection. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
11. Miller, S.P., Cofer, D.D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics (2009) (submitted for publication)
12. Ölveczky, P.C., Boronat, A., Meseguer, J., Pek, E.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude (2010), `http://www.ifi.uio.no/RealTimeMaude/AADL/`
13. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. Electronic Notes in Theoretical Computer Science 176(4), 5–27 (2007)

14. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
15. SAE AADL Team: AADL homepage (2009), http://www.aadl.info/
16. Sokolsky, O., Lee, I., Clarke, D.: Process-algebraic interpretation of AADL models. In: Kordon, F., Kermarrec, Y. (eds.) Reliable Software Technologies – Ada-Europe 2009. LNCS, vol. 5570, pp. 222–236. Springer, Heidelberg (2009)
17. Yang, Z., Hu, K., Ma, D., Pi, L.: Towards a formal semantics for the AADL behavior annex. In: Proc. DATE 2009. IEEE, Los Alamitos (2009)

Testing Probabilistic Distributed Systems*

Robert M. Hierons[1] and Manuel Núñez[2]

[1] Department of Information Systems and Computing, Brunel University
Uxbridge, Middlesex, UB8 3PH United Kingdom
rob.hierons@brunel.ac.uk

[2] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Madrid, Spain
mn@sip.ucm.es

Abstract. There has been much interest in the testing of systems that have physically distributed interfaces and this has been encouraged by recent trends towards the use of such systems. Most formal work in this area has considered the testing of deterministic systems based on deterministic models. However, distributed systems are usually nondeterministic and often can be seen as probabilistic systems in which required or expected probabilities can be attached to the allowable events. This paper provides a formal testing framework for systems with physically distributed interfaces where nondeterministic decisions among alternatives are probabilistically quantified. It first considers testing from systems where there is a unique type of action. In this setting, a *generative* interpretation of probabilities is adequate and a formal framework to test these systems is provided. However, the observable events of a system are usually divided into inputs and outputs. In such situations it is necessary to use the *reactive* interpretation of probabilities.

1 Introduction

It is widely accepted that testing is an important part of the software development process and manual testing is typically expensive and error prone. This has led to interest in the testing of systems based on specifications or models, an area usually called *model based testing* (MBT). Most approaches to MBT use state-based models such as finite state machines or labelled transition systems (see, for example, [18,22,4,11,9]). In addition to providing a formal framework to reason about the correctness of systems, MBT has the enormous advantage of facilitating the automation of the main testing tasks (see [27] for a review of some tool-supported theories).

In order to develop and apply MBT techniques it is necessary to formally define what it means for an implementation to conform to a specification or

* This research was partially supported by the Spanish MEC projects WEST/FAST (TIN2006-15578-C02-01) and TESIS (TIN2009-14312-C02-01), the UK EPSRC project Testing of Probabilistic and Stochastic Systems (EP/G032572/1) and the UCM-BSCH programme to fund research groups (GR58/08 - group number 910606).

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 63–77, 2010.
© IFIP International Federation for Information Processing 2010

model. Such definitions are usually called *implementation relations*. For a given model, an implementation relation defines exactly which implementations are correct and thus testing is based on both the model and the implementation relation. Many implementation relations are either trace inclusion or extensions of this such as ioco [26].

In this paper we assume that the *system under test* (SUT) has physically distributed interfaces, called *observation ports* or just ports, and so that in testing we place one tester at each observation port. We assume that the testers do not directly communicate with one another during testing and also that this corresponds to the situation in use: there will be separate users or systems at the ports and while these may share information later, they do not synchronise their actions through communicating when using the SUT. The restrictions imposed on testing have the benefit of making the test infrastructure easier and cheaper to implement: there is no need to introduce an external communications network between the testers. We make one additional assumption, usually made in this context, which is that the testers do not have access to a global clock. It is known that this *decentralised* approach reduces test effectiveness and this topic has received much attention (see, for example, [24,20,23,16]).

The decentralised approach to testing is simpler to implement since there is no need to produce an external communications network through which the testers synchronise their actions. In addition, while we can test individual components without this (assuming each has at most one interface), there is still a need to test the entire system. Crucially, if the decentralised approach corresponds to the *expected use* of the system then the reduced observational power, and corresponding implementation relations, capture what it means for the SUT to be a correct implementation. Thus, even if we can synchronise the testers there is a need to define implementation relations that capture the observational power of agents that will interact with the SUT in use. There can be at least two negative consequences of not doing this. First, our testing may be inefficient: we might produce test cases that aim to find differences between the behaviour of the SUT and the specification where these differences cannot be observed in use and thus are not faults. Second, our testing might not be sound since we could declare that the SUT fails a test even though the behaviour was not observationally different from a behaviour in the specification.

Previous research on testing systems with physically distributed interfaces has not considered models with probabilities. However, distributed systems are often probabilistic in nature and this has led to significant interest in probabilistic models and to the study of semantic models for probabilistic processes. If we consider only probabilistic extensions based on testing we can mention [6,17,29,25,7,21,19,5,8,12]. Even though there are plenty of proposals to test a wide variety of probabilistic processes, as far as we are aware there has been no previous work on the problem of testing with physically distributed interfaces where nondeterministic decisions are probabilistically quantified.

This paper considers two approaches to adding probabilities. First we consider testing from labelled transition systems in which we apply a *generative approach*

[10]; the probabilities of the transitions leaving a state sum to 1. However, to properly specify most systems it is necessary to distinguish between inputs and outputs. The system determines which outputs are produced while the environment controls the inputs. For such systems the generative approach corresponds to having probabilistic information about the environment in addition to the required behaviours of the SUT. Thus, for systems with a distinction between inputs and outputs we use a combination of the *reactive* [17] and generative approaches. Our approach is reactive for inputs: given state s and input $?i$, the sum of the probabilities of the transitions leaving s with input $?i$ is 1. However, it is generative for outputs: given state s, the sum of the probabilities of the transitions leaving s and labelled by an output is 1. Intuitively, this mixed approach corresponds to having probabilistic requirements for the SUT but not placing restrictions on the behaviour of the environment. Similar ideas are used in the probabilistic models considered in [28,2].

In analysing a model we determine the probability of making particular observations. Interestingly, it transpires that this can be problematic when we distinguish between inputs and outputs as a result of races. Specifically, observations are not global traces of the system but sets of global traces that are indistinguishable when there are independent agents/testers at the ports. There can be races between events at different ports and where one or more of these events are inputs the reactive-generative setting does not provide probabilistic information regarding the outcome of such races. As a result, we outlaw these types of races and provide a condition under which such races cannot occur.

It is worth mentioning that our implementation relations are conservative extensions of previous notions for the non-distributed and/or non-probabilistic framework. For the generative approach we show that if we have only one port and we forget probabilistic information then our implementation relation is equivalent to trace inclusion. If we allow probabilities then our implementation relation is equivalent to the natural extension of trace.

This paper is structured as follows. In Section 2 we give preliminary definitions regarding observations that can be made when testing distributed systems. Section 3 then considers the problem of testing from probabilistic labelled transition systems. In Section 4 we investigate the problem of testing from a probabilistic system having a distinction between inputs and outputs. Finally, in Section 5, we draw conclusions and discuss future work.

2 Preliminaries

Throughout this paper we assume that there are m observation ports and that we identify these using the integers in $\mathcal{O} = \{1, \ldots, m\}$. If $\mathcal{A}ct$ denotes the set of actions then this is partitioned into sets $\mathcal{A}ct_1, \ldots, \mathcal{A}ct_m$: for all $o \in \mathcal{O}$, $\mathcal{A}ct_o$ denotes the set of actions that can be observed at o. In Section 4 we consider input output transition systems. When testing from an input output transition system it is common to assume that we can observe the system being in a stable (quiescent) state, in which it cannot progress without receiving further input, and this observation is denoted δ. Quiescence can be observed at all ports.

When a system interacts with its environment it does so through a sequence of actions in $\mathcal{A}ct \cup \{\delta\}$ and this is called a *global trace*. Given a global trace $\sigma \in (\mathcal{A}ct \cup \{\delta\})^*$ we can define the projection $\pi_o(\sigma)$ of σ onto port o, and this is called a *local trace*, in the following way (ϵ represents the empty sequence):

1. $\pi_o(\epsilon) = \epsilon$.
2. If $z \in \mathcal{A}ct_o \cup \{\delta\}$ then $\pi_o(z\sigma) = z\pi_o(\sigma)$.
3. If $z \notin \mathcal{A}ct_o \cup \{\delta\}$ then $\pi_o(z\sigma) = \pi_o(\sigma)$.

Consider, for example, a global trace $a_1b_2c_1$ in which a_1 and c_1 are at port 1 and b_2 is at port 2. Then $\pi_1(a_1b_2c_1) = a_1c_1$ and $\pi_2(a_1b_2c_1) = b_2$.

As stated above, we partition the set of actions so that each can occur at only one port. For systems where the same event can occur at two or more ports we can simply label an event with the port number.

Given global traces $\sigma, \sigma' \in (\mathcal{A}ct \cup \{\delta\})^*$ and observation port o we write $\sigma \sim_o \sigma'$ if σ and σ' cannot be distinguished when only observing the local traces at o. More formally, $\sigma \sim_o \sigma'$ if and only if $\pi_o(\sigma) = \pi_o(\sigma')$. For example, if a_1 and c_1 are events at port 1 and b_2 is an event at port 2 then we have that $a_1b_2c_1 \sim_1 a_1c_1 \sim_1 b_2a_1b_2c_1b_2$. However, a_1c_1 and c_1a_1 are not related under $\sim_1$. We can strengthen this to say what it means for two global traces to be indistinguishable when observing the local traces. Given global traces $\sigma, \sigma' \in (\mathcal{A}ct \cup \{\delta\})^*$ we write $\sigma \sim \sigma'$ if σ and σ' cannot be distinguished when only observing the local traces. Thus, $\sigma \sim \sigma'$ if and only if for all $o \in \mathcal{O}$ we have that $\pi_o(\sigma) = \pi_o(\sigma')$. For example, we have that $a_1b_2c_1 \sim b_2a_1c_1$ since $\pi_1(a_1b_2c_1) = a_1c_1 = \pi_1(b_2a_1c_1)$ and $\pi_2(a_1b_2c_1) = b_2 = \pi_2(b_2a_1c_1)$.

Relations $\sim$ and $\sim_o$ are equivalence relations and so define equivalence classes. Given global trace σ and port o we let $[\sigma]_o$ denote the equivalence class of σ with respect to $\sim_o$ and this is the set of global traces that are indistinguishable from σ when only observing the local trace at o. Thus,

$$[\sigma]_o = \{\sigma' \in (\mathcal{A}ct \cup \{\delta\})^* | \pi_o(\sigma') = \pi_o(\sigma)\}$$

Similarly, given global trace σ we let $[\sigma]$ denote the equivalence class of σ with respect to $\sim$ and this is the set of global traces that are indistinguishable from σ when only observing the local traces. Thus,

$$[\sigma] = \{\sigma' \in (\mathcal{A}ct \cup \{\delta\})^* | \forall o \in \mathcal{O} : \pi_o(\sigma') = \pi_o(\sigma)\}$$

Clearly we have that $\sigma' \sim \sigma$ if and only if for all $o \in \mathcal{O}$ we have that $\sigma' \sim_o \sigma$. In addition, for all σ we have that $[\sigma] = \bigcap_{o \in \mathcal{O}} [\sigma]_o$.

In this paper the set $(0, 1]$ denotes all non-zero probabilities; all real numbers that are greater than 0 and no larger than 1. In addition, $[0, 1] = \{0\} \cup (0, 1]$. In general, we will use multisets of probabilities, instead of sets, since the same probability can be associated with different transitions that we are somehow counting together. We use $\{\!|$ and $|\!\}$ as the delimiters for multisets.

3 Implementation Relations for Labelled Transition Systems: A Purely Generative Approach

Labelled transitions systems (LTSs) have states, actions and transitions between states. Recent work has shown how we can define such systems with multiple ports[1] (see, for example, [14,13]). Under the generative interpretation of probabilities, in a state q there is a set of possible transitions, each transition has a probability and the probabilities sum to 1.

Definition 1. *A* probabilistic labelled transition system (PLTS) *s is defined by a tuple $s = (Q, \mathcal{A}ct, T, q_{in})$ in which Q is a countable set of states, $q_{in} \in Q$ is the initial state, $\mathcal{A}ct$ is a countable set of actions, and $T \subseteq Q \times \mathcal{A}ct \times Q \times (0, 1]$, is the transition relation. A transition (q, a, q', p) means that when in state q, with probability p the next event moves s to state q' with action $a \in \mathcal{A}ct$. Naturally, we cannot have two transitions $(q, a, q', p) \in T$ and $(q, a, q', p') \in T$ in which $p \neq p'$. The set $\mathcal{A}ct$ of actions is partitioned into subsets $\mathcal{A}ct_1, \ldots, \mathcal{A}ct_m$ where for all $o \in \mathcal{O}$ we have that $\mathcal{A}ct_o$ denotes the set of actions that can occur at observation port o. We require that for every state $q \in Q$ either $\sum \{\!| \, p \mid \exists a, q' : (q, a, q', p) \in T \, |\!\}$ is equal to 1 or q is a deadlock state and so this sum is equal to zero. We let $\mathcal{PLTS}(\mathcal{A}ct)$ denote the set of PLTSs with action set $\mathcal{A}ct$.*

We say that the process s is finitely branching *if for every state $q \in Q$ there are only a finite number of transitions with starting state q. In this paper we only consider processes that are finitely branching.*

Any state $q \in Q$ induces an LTS derived from s by setting the initial state to q, that is, abusing the notation we consider $q = (Q, \mathcal{A}ct, T, q)$.

Let us note that all transitions have non-zero probability. An alternative is to allow the probability of a transition to be from the set $[0, 1]$ but we can simply delete any transition with probability 0 since it does not affect the behaviour of the PLTS. We now introduce notation for PLTSs that we will use in defining implementation relations. In particular, we define the probability of making a sequence of observations from a state q of a PLTS s.

Definition 2. *Given a PLTS $s = (Q, \mathcal{A}ct, T, q_{in})$, a state q of s, and $\sigma \in \mathcal{A}ct^*$, we let $prob(q, \sigma)$ denote the probability of performing the sequence σ from state q. Formally,*

$$prob(q, \sigma) = \begin{cases} 1 & \text{if } \sigma = \epsilon \\ \sum \{\!| \, p \cdot prob(q', \sigma') \mid (q, a, q', p) \in T \, |\!\} & \text{if } \sigma = a\sigma' \end{cases}$$

We say that $\sigma \in \mathcal{A}ct^*$ is a trace *of s if $prob(q_{in}, \sigma) > 0$. We denote by $L(s)$ the set of traces of s.*

[1] The work actually defines input output transition systems with multiple ports but it is straightforward to adapt this approach.

We will define an implementation relation for the generative case in the distributed setting. Essentially we wish to say that every $\sigma \in Act^*$ that can be observed in the specification must have the same probability of occurrence in the SUT. However, as a result of there being multiple ports, there may be alternative global traces of the specification and of the implementation that are indistinguishable. We therefore compare the probability of observing elements of equivalence classes rather than particular global traces.

Definition 3. *Let $s = (Q, I, O, T, q_{in})$ be a PLTS and $\sigma \in Act^*$. We define the* probability with which s performs the equivalence class $[\sigma]$, *denoted by $prob(s, [\sigma])$, as*

$$\sum \{\!\!\{ \, prob(q_{in}, \sigma') \mid \sigma' \in [\sigma] \, \}\!\!\}$$

Let s, r be PLTSs. We write $r \sqsubseteq^G s$ if for all $\sigma \in L(s)$ we have that $prob(s, [\sigma]) = prob(r, [\sigma])$.

In order to show that our relation is a conservative extension of the *classical* non-distributed and non-probabilistic framework, we have the following result where probabilistic information can be reduced to trace containment.

Proposition 1. *Let s, r be single-port PLTSs. We have $r \sqsubseteq^G s$ implies $L(r) \supseteq L(s)$.*

Proof. The proof is easy by taking into account that, under the specified conditions, if $s = (Q, I, O, T, q_{in})$ then for all $\sigma \in \mathcal{A}ct^*$ we have $prob(s, [\sigma]) = prob(q_{in}, \sigma)$. □

However, if we retain probabilistic information we find that our relation is equivalent to the natural extension of trace inclusion.

Proposition 2. *Let s, r be single-port PLTSs. We have $r \sqsubseteq^G s$ implies that for all $\sigma \in L(s)$ we have that $prob(r, \sigma) = prob(s, \sigma)$.*

Proof. Since $r \sqsubseteq^G s$, if $\sigma \in L(s)$ then we have that $prob(s, [\sigma]) = prob(r, [\sigma])$. However, for all $\sigma \in \mathcal{A}ct^*$ we have $prob(s, [\sigma]) = prob(s, \sigma)$ and $prob(r, [\sigma]) = prob(r, \sigma)$ and so the result follows. □

We would now like to comment on a limitation of $\sqsubseteq^G$. Even though it seems to be an appropriate adaptation of the generative approach to the distributed setting, it has a main drawback: it does not properly capture the notion of *complete trace*. This problem is not related to the additional complexity introduced by distributed ports. Therefore, we will illustrate it with two single-port systems. Let us consider the following two processes shown in Figure 1:

1. Process s initially produces a with two different transitions. The first transition, with probability $\frac{1}{2}$, reaches a deadlock state while the second transition, also with probability $\frac{1}{2}$, reaches a state that can perform b and reaches a deadlock state.

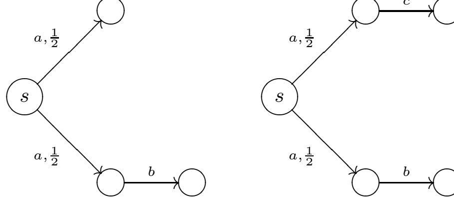


Fig. 1. Processes s and r

2. Process r initially produces a with two different transitions. The first transition, with probability $\frac{1}{2}$, reaches a state that can perform c followed by deadlock while the second transition, also with probability $\frac{1}{2}$, reaches a states that can perform b and then is also followed by deadlock.

We certainly have that $r \sqsubseteq^G s$ since we only check the probabilities associated with the traces a (equal to 1 in both processes) and ab (equal to $\frac{1}{2}$ in both processes). However, we might consider that s is offering a behaviour that r cannot *simulate*. Specifically, s can perform a to reach a deadlock state while r cannot. In order to capture this type of behaviour, we can extend the alphabet of actions with a special action that will be associated with deadlock states. We will add self-loops labelled by this special action and probability 1 in deadlock states. This is exactly the idea of *quiescence* that we will use in the next section.

Definition 4. *Let $s = (Q, \mathcal{A}ct, T, q_{in})$ be a PLTS. We can extend the set of transitions T to a new set T_δ by adding the transition $(q, \delta, q, 1)$ for each state $q \in Q$ such that $\sum \{\!| \, p \mid \exists a, q' : (q, a, q', p) \in T \, |\!\} = 0$. The* augmented *PLTS is given by $s_\delta = (Q, \mathcal{A}ct \cup \{\delta\}, T_\delta, q_{in})$.*

Let s, r be PLTSs and s_δ, r_δ be the corresponding augmented PLTSs. We write $r \sqsubseteq^G_\delta s$ if for all $\sigma \in L(s_\delta)$ we have that $prob(s_\delta, [\sigma]) = prob(r_\delta, [\sigma])$.

This new implementation relation properly captures complete traces since a trace σ reaching a deadlock state is *transformed* into the traces $\sigma\delta^n$, for $n \geq 0$. For example, if we consider the processes r and s that we used to motivate the new relation we have that they are not related under $\sqsubseteq^G_\delta$. In addition to cope with complete traces, the new implementation relation has a very interesting property. We have that the asymmetry in its definition, since only traces of s are considered, is not real since this order relation masks an equivalence relation as the following result shows.

Proposition 3. *Let s, r be PLTSs. We have that $r \sqsubseteq^G_\delta s$ implies $s \sqsubseteq^G_\delta r$.*

Proof. We use proof by contradiction, assuming that $r \sqsubseteq^G_\delta s$ but that $s \sqsubseteq^G_\delta r$ does not hold. Let σ be a shortest element of $L(r)$ such that $prob(r, [\sigma]) \neq prob(s, [\sigma])$. If $prob(s, \sigma) > 0$ then, since $prob(r, [\sigma]) \neq prob(s, [\sigma])$, we know that $r \sqsubseteq^G_\delta s$ does not hold, providing a contradiction. We therefore must have that $prob(s, \sigma) = 0$ and therefore $\sigma \notin L(s)$.

Let k denote the length of $\sigma \in \mathcal{A}ct^*$ and assume that in s we have l distinct equivalence classes $[\sigma_1], \ldots, [\sigma_l]$ of sequences that have length k. Let us remark that traces σ reaching a deadlock state and having a length $k_\sigma < k$ will contribute to this set of classes with the class $[\sigma\delta^{k-k_\sigma}]$.

Since probabilities are generative, it is easy to check that $\sum_{1 \leq i \leq l} prob(s, [\sigma_i]) = 1$. Further, since $r \sqsubseteq_\delta^G s$, $\sum_{1 \leq i \leq l} prob(r, [\sigma_i]) \geq \sum_{1 \leq i \leq l} prob(s, [\sigma_i])$ and so $\sum_{1 \leq i \leq l} prob(r, [\sigma_i]) = 1$.

Let us note that $prob(r, [\sigma]) \neq prob(s, [\sigma])$ and $prob(s, \sigma) = 0$ and so we have $\sum_{1 \leq i \leq l} prob(r, [\sigma_i]) + prob(r, [\sigma]) > 1$. However, for all $1 \leq i \leq l$ we have that $\sigma \not\sim \sigma_i$ and the sequences $\sigma, \sigma_1, \ldots, \sigma_l$ all have length k and so we must have that $\sum_{1 \leq i \leq l} prob(r, [\sigma_i]) + prob(r, [\sigma]) \leq 1$. This provides a contradiction and so the result follows. □

Thus, $\sqsubseteq_\delta^G$ is an equivalence relation. Normally, an implementation relation is a preorder but not an equivalence relation since it allows a range of implementation decisions. Future work will consider whether there are suitable implementation relations for the generative case that are not equivalence relations.

4 Implementation Relations for Input Output Transition Systems: A Reactive-Generative Approach

Many systems interact with their environment through inputs and outputs and in this section we consider such systems and the observations that can be made, which are sequences of inputs and output (input output sequences).

When a system interacts with its environment through inputs and outputs there is often an asymmetry between these since the environment controls the inputs while the system controls the outputs. This has led to the use of input output transition systems (IOTSs), which essentially are LTSs where we distinguish between input and output. We now define a probabilistic IOTS that has multiple ports. We use the reactive scenario for inputs and the generative for outputs. However, we do attach probabilities to inputs since there may be more than one transition leaving a state q with a given input $?x$: the environment chooses the input to supply but the system determines which transition to take.

Definition 5. *A* probabilistic input-output transition system (PIOTS) *s is defined by a tuple $s = (Q, I, O, T, q_{in})$ in which Q is a countable set of states, $q_{in} \in Q$ is the initial state, I is a countable set of inputs, O is a countable set of outputs, and $T \subseteq Q \times (I \cup O) \times Q \times (0, 1]$ is the transition relation. A transition (q, a, q', p) means that from state q it is possible to move to state q' with action $a \in I \cup O$ with probability p. Again, we cannot have two transitions $(q, a, q', p) \in T$ and $(q, a, q', p') \in T$ in which $p \neq p'$. If $a \in O$ then we should interpret the probability p of (q, a, q', p) as meaning that if an output occurs in state q before input is provided then with probability p this transition occurs. Therefore, for every state q we must have that $\sum \{\!| \, p \mid \exists q', a : (q, a, q', p) \in T \wedge a \in O \, |\!\}$ is either 1 or 0 (if the state cannot produce any output). Further, if $a \in I$ then*

we must have that the sum of the probabilities of transitions leaving q with input a, that is $\sum \{\!| p \mid \exists q' : (q, a, q', p) \in T |\!\}$, is either 1 or 0 (if the input is not available at that state). This means that once an available input is chosen by the environment, we can forget the other available inputs and concentrate on the probability distribution function governing the transitions labelled by a.

A state $q \in Q$ is quiescent *if all transitions from q involve input. We can extend the set of transitions T to a new set T_δ by adding the transition $(q, \delta, q, 1)$ for each quiescent state q. We let $\mathcal{A}ct = I \cup O \cup \{\delta\}$ denote the set of actions.*

We partition the set I of inputs into $I_1, \ldots, I_m$ in which for port $o \in \mathcal{O}$ we have that I_o is the set of inputs that can be received at port o. Similarly, we partition the set O of outputs into sets $O_1, \ldots, O_m$. We let $\mathcal{PIOTS}(I, O)$ denote the set of PIOTSs with input set I and output set O.

We say that the process s is input-enabled *if for all $q \in Q$ and all $?i \in I$ there exists $q' \in Q$ and probability $p \in (0, 1]$ such that $(q, ?i, q', p) \in T$. We say that the process s is* output-divergent *if it can reach a state in which there is an infinite path that contains outputs only. We say that s is* finitely branching *if for every state $q \in Q$ there are only a finite number of transitions with starting state q. In this paper we only consider processes that are finitely branching and are not output-divergent.*

Let us remark that if we consider δ as a regular output action, then we can say that the sum of the probabilities associated with a state is always equal to 1, that is, $\sum \{\!| p \mid \exists q', a : (q, a, q', p) \in T_\delta \wedge a \notin I |\!\} = 1$. In addition, as usual, we precede the name of an input by ? and we precede the name of an output by !. We will often label inputs and outputs in order to make their port clear. For example, $?i_p$ denotes an input at p and $!o_p$ denotes an output at p. An alternative, used in [14], to this notion of (probabilistic) input output transition systems is to allow outputs to be tuples of values but the formalism used in this paper has the advantage of simplifying the notation and analysis.

Traces are sequences of actions, possibly including quiescence, and are usually called *suspension traces*. In this paper we simply call them global traces. The following is standard notation in the context of **ioco**: the implementation relation usually used in testing from a single-port IOTS [26].

Definition 6. *Let $s = (Q, I, O, T, q_{in})$ be a PIOTS. We use the following notation.*

1. *If $(q, a, q', p) \in T_\delta$, for $a \in \mathcal{A}ct$, then we write $q \xrightarrow{a} q'$ and $q \xrightarrow{a}$.*
2. *We write $q \stackrel{\sigma}{\Longrightarrow} q'$ for $\sigma = a_1 \ldots a_m \in \mathcal{A}ct^*$ if there exist $q_0, \ldots, q_m$, $q = q_0$, $q' = q_m$ such that for all $0 \leq i < m$ we have that $q_i \xrightarrow{a_{i+1}} q_{i+1}$.*
3. *If there exists q' such that $q_{in} \stackrel{\sigma}{\Longrightarrow} q'$ we say that σ is a* trace *of s. We let $\mathcal{T}r^*(s)$ denote the set of traces of s.*
4. *Let $q \in Q$ be a state and $\sigma \in \mathcal{T}r^*(s)$ be a trace. We introduce the following concepts.*
   (a) *s **after** $\sigma = \{q \in Q | q_{in} \stackrel{\sigma}{\Longrightarrow} q\}$.*
   (b) ***out**$(q) = \{!o \in O | q \stackrel{!o}{\Longrightarrow}\}$.*

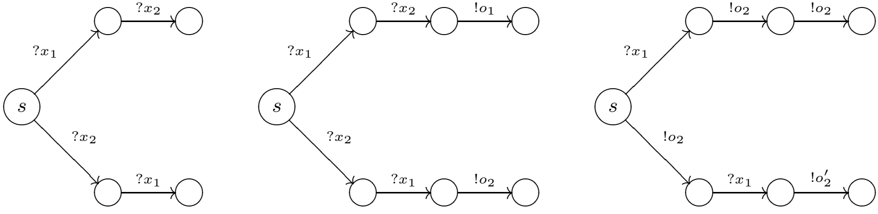


Fig. 2. Process s_1, s_2, and s_3

5. *Given an input output sequence σ, we let* $\mathbf{in}(\sigma)$ *denote the input sequence formed by removing all of elements of σ that are not inputs.*

Let us note that we have initially abstracted probabilistic information in the definition of trace. This information will be incorporated when defining our implementation relations.

Next we recall the definition of **ioco** [26] and its adaption to systems with multiple ports that we call **dioco** [14].

Definition 7. *Let $s, r \in \mathcal{PIOTS}(I, O)$. We write r* **ioco** *s if for every trace $\sigma \in \mathcal{T}r^*(s)$ we have that* $\mathbf{out}(r \textbf{ after } \sigma) \subseteq \mathbf{out}(s \textbf{ after } \sigma)$.

We write r **dioco** *s if for every trace σ such that $r \stackrel{\sigma}{\Longrightarrow} r'$ for some r' that is in a quiescent state, if there is a global trace $\sigma_1 \in \mathcal{T}r^*(s)$ such that $\mathbf{in}(\sigma_1) \sim \mathbf{in}(\sigma)$ then there exists a trace $\sigma' \in \mathcal{T}r^*(s)$ such that $s \stackrel{\sigma'}{\Longrightarrow}$ and $\sigma' \sim \sigma$.*

As expected, the previous implementation relation discarded probabilistic information. It is important to explain why we restrict attention to traces that end in quiescent states. Let us consider the following processes:

1. Process s that can produce output $!o_1$ at port 1, then output $!o_2$ at port 2 and then deadlocks; and
2. Process r that can produce output $!o_2$ at port 2, then output $!o_1$ at port 1 and then deadlocks.

If we can observe just $!o_2$ in r then we can distinguish between r and s but the environment cannot block the following $!o_1$. This is because the agents/testers at the ports can only exchange information in quiescent states as a result of not being able to synchronise their actions during testing.

An interesting, but more complicated, alternative has been recently defined in [15] where only infinite traces are used to define **dioco**. Since a finite trace is a prefixes of an infinite one (since we can always extend a trace with either an output or with δ), this notion captures the nature of **dioco**. For example, the infinite traces that the previous processes can perform are $!o_1!o_2\delta\delta\cdots$ and $!o_2!o_1\delta\delta\cdots$, respectively, and these are related by $\sim$.

In order to define an appropriate probabilistic extension of **dioco**, we need to restrict our attention to a class of systems without *pathological* behaviours. Let

us consider the process s_1 in Figure 2 left, in which $?x_1$ is input at port 1 and $?x_2$ is input at port 2. All transitions have probability 1. We might ask what probability we should associate with global traces that are indistinguishable from $?x_1?x_2$. We cannot simply sum the probabilities of the two global traces that are equivalent to $?x_1?x_2$ under $\sim$ since we obtain the result 2.

Now let us consider process s_2 in Figure 2 centre, in which $!o_1$ is at port 1 and $!o_2$ is at port 2. We might ask what probability we should associate with global traces that are indistinguishable from $?x_1?x_2!o_1$. However, whether we observe $?x_1?x_2!o_1$ depends on the outcome of a race between $?x_1$ and $?x_2$ and we have no probabilistic information regarding this race. Thus, we cannot give such a probability. However, for a model to be useful it should define the probability of an observation. Thus, we either need to use a generative approach, which provides the probabilities for the outcomes of a race, or we restrict attention to PIOTSs without such races. We now discuss how the latter can be done.

Previous work on using IOTSs for distributed systems has defined the *mioco* implementation relation where we make global observations but there are multiple ports [3]. This does not require that an IOTS is input-enabled but does insist that if no transition is defined for input $?i_o \in I_o$ in state q then there are no transitions from q with an input from I_o. This corresponds to the SUT being able to block input at an interface. We make a similar assumption, which is that for any state q we cannot have transitions from q for inputs from different ports. This corresponds to a design that avoids races by restricting input to one port at a time. This restriction is motivated by the observation that, as seen in Figure 2 left and centre, we need to avoid such races if we wish to be able to assign probabilities to observations. Interestingly, work on Message Sequence Charts (MSCs) [1] has defined a pathology in which the next events after branching are on different processes: our restriction is similar to outlawing this pathology.

It is not sufficient to outlaw races between inputs since we can have a race between an input and an output at different ports. An example of this is given in s_3 from Figure 2 right. Here we cannot assign a probability to there being a global trace indistinguishable from $?x_1!o_2!o_2$ since there is a race between $?x_1$ and $!o_2$. However, having different possible outputs at different ports or inputs and outputs at the same port, from a state q, causes no problem since PIOTSs do contain probabilistic information regarding these choices. Thus, a PIOTS in which there can be races between inputs or between inputs and outputs should not be allowed since we cannot assign probabilities to traces. We will require any PIOTS to be consistent, as defined below.

Definition 8. *Let $s = (Q, I, O, T, q_{in})$ be a PIOTS. We say that s is* consistent *if for every state $q \in Q$ if there exist $a_1, a_2 \in I \cup O$ such that $q \xrightarrow{a_1}$ and $q \xrightarrow{a_2}$ then either both of them are outputs or they are at the same port.*

Throughout the rest of this paper we only consider PIOTSs that are consistent, in addition to being finitely branching and not output-divergent. We can adapt notation defined for PLTSs; we consider a PIOTS to be a PLTS and so reuse concepts such as the probability $prob(q, \sigma)$ of performing a given sequence of

actions σ from a state q. We assume that observations are made in quiescent states and restrict attention to paths that end in quiescent states and adapt the **dioco** implementation relation to the probabilistic framework.

Definition 9. *Let $s, r \in \mathcal{PIOTS}(I, O)$. We write $r \sqsubseteq^R s$ if for every global trace σ such that $s \stackrel{\sigma}{\Longrightarrow} s'$ for some s' that is in a quiescent state, we have that $prob(s, [\sigma\delta]) = prob(r, [\sigma\delta])$.*

In the above we include δ in the global traces over which we sum probabilities to ensure that we are considering quiescent traces in each process. The notation $prob(s, [\sigma\delta])$ is given in Definition 3. Let us remark that, in contrast with $\sqsubseteq^G_\delta$ but similar to $\sqsubseteq^G$, the relation $\sqsubseteq^R$ is not symmetric.

Proposition 4. *There exist PIOTSs r and s such that $r \sqsubseteq^R s$ but we do not have that $s \sqsubseteq^R r$.*

Proof. It is sufficient to consider processes r and s such that:

1. Process s has only one transition which is a self-loop with label δ.
2. Process r has a self-loop with label δ and a transition with input $?x$ to a state r' that has only one transition, which is a self-loop with label δ.

Then $r \sqsubseteq^R s$ since the quiescent traces of s contain only δ and these are also traces of r and have the same probabilities. However, to see that $s \sqsubseteq^R r$ does not hold it is sufficient to consider the quiescent trace $?x\delta$ of r. □

If we consider single-port systems and we also forget probabilistic information then we have that our relation reduces to **ioco**. Let us remark that **ioco** is only defined for input-enabled implementations and so we have to make this restriction. However, input-enabled single-port PIOTSs can be still consistent since, obviously, they cannot perform actions at different ports.

Proposition 5. *Let s, r be single-port PIOTSs such that r is input-enabled and $r \sqsubseteq^R s$. If $\sigma \in \mathcal{T}r^*(s)$ and either $a \in$ **out**$(r$ **after** $\sigma)$ or $a \in$ **out**$(s$ **after** $\sigma)$ then $prob(r, \sigma a) = prob(s, \sigma a)$.*

Proof. We will use proof by induction on the length of σ. The base case is $\sigma = \epsilon$. First, since s is not output-divergent, the sum of the probabilities of quiescent traces of s that contain only output and exactly one δ is 1. In addition, there are a finite number of such sequences in s (since s is finitely-branching and is not output-divergent), and since $r \sqsubseteq^R s$ and there is only one port we have that these are also quiescent sequences of r and in r they have the same probabilities. It is now sufficient, for each element $a \in$ **out**$(r$ **after** $\epsilon) \cup$ **out**$(s$ **after** $\epsilon)$, to consider such traces that start with output a and sum the probabilities of the corresponding traces. The result thus holds for the base case.

Now we assume that the result holds for all traces of length less than that of $\sigma \in \mathcal{T}r^*(s)$. By the inductive hypothesis, since every prefix of σ is a trace of s we have that $prob(s, \sigma) = prob(r, \sigma)$. Since s is finitely-branching we have that **out**$(s$ **after** $\sigma)$ is finite and let us suppose that **out**$(s$ **after** $\sigma) = \{a_1, \ldots, a_k\}$.

Since s is not output-divergent, for each $1 \leq i \leq k$ we have that there is a finite set A_i of (minimal) traces of s that extend σ with only outputs and δ and that contain exactly one extra δ (at the end). Thus, σ' is in A_i if and only if $\sigma' \in \mathcal{T}r^*(s)$, σ' extends σ with outputs and exactly one δ, ends in δ, and σa_i is a prefix of σ'. Since outputs are generative, $prob(s, \sigma a_i) = \sum_{\sigma' \in A_i} prob(s, \sigma')$. Since $r \sqsubseteq^R s$ and there is only one port, for all $\sigma'' \in A_i$ we must have that $prob(r, \sigma'') = prob(s, \sigma'')$ and so for all $1 \leq i \leq k$ we have that $prob(r, \sigma a_i) \geq prob(s, \sigma a_i)$. Now let us note that we have $prob(s, \sigma) = prob(r, \sigma)$, $prob(s, \sigma) = \sum_{1 \leq i \leq k} prob(s, \sigma a_i)$ and $prob(r, \sigma) \geq \sum_{1 \leq i \leq k} prob(r, \sigma a_i) = \sum_{1 \leq i \leq k} prob(s, \sigma a_i)$. Thus, for all $1 \leq i \leq k$ we have $prob(r, \sigma a_i) = prob(s, \sigma a_i)$ and $\mathbf{out}(r \ \mathbf{after} \ \sigma) = \{a_1, \ldots . a_k\}$. □

The following is an immediate consequence.

Proposition 6. *Let s, r be single-port PIOTSs. If r is input-enabled then we have $r \sqsubseteq^R s$ implies that r* **ioco** *s.*

Our last result relates the implementation relations presented in this paper. If the set of inputs is empty then the relations should coincide. Under this assumption we do not have $r \sqsubseteq^R s$ if and only if $r \sqsubseteq^G s$ due to the inadequate treatment of complete traces under $\sqsubseteq^G$. Fortunately, the expected result holds with the variant of $\sqsubseteq^G$.

Proposition 7. *Let s, r be single-port PIOTSs with empty sets of inputs. We have $r \sqsubseteq^R s$ if and only if $r \sqsubseteq^G_\delta s$.*

5 Conclusions

This paper has investigated the problem of testing a probabilistic system that has distributed interfaces, called ports. This is a significant problem since distributed systems are becoming increasingly important and are often probabilistic in nature. Interestingly, while there has been much separate work on testing probabilistic systems and testing systems with distributed interfaces, this appears to be the first that considers the combination.

We assume that the *system under test (SUT)* has physically distributed ports and we place one local tester at each port. We also assume that the testers do not communicate with one another during testing and this corresponds to the situation in use: the separate users/systems at the ports might share information later but do not synchronise when using the SUT. Each local tester observes a projection of the global trace that occurs and this is called a local trace.

Initially we considered a generative situation where the probabilities on transitions leaving a state sum to 1. This is similar to a Markov Chain and corresponds to testing a closed system or one in which we know the probabilities of events from the environment that affect the SUT. In this context we explored testing from probabilistic labelled transition systems by defining an implementation relation where the local testers may return their sets of observations and a verdict can be produced from these (local traces).

Sometimes we distinguish between input and output: the SUT controls output and the environment controls input. Here, input output transition systems are more suitable that labelled transition systems and we require a reactive scenario. We investigated the problem of testing from a probabilistic input output transition system (PIOTS) that has multiple ports. In analysing such a model we need to determine the probability of observing a trace that is equivalent to a given trace σ. For some models the presence of races means that this probability is undefined and so we defined a class of model for which such problems do not occur and we also defined an implementation relation.

There are several avenues for future work. First, there is the problem of producing test generation algorithms that direct testing in order to achieve a given objective. It is also necessary to consider how tests should be applied and verdicts assigned but the latter essentially corresponds to estimating the probabilities of sequences of events in the SUT through sampling and comparing the estimate with the required probability. In addition, simulation relations have been defined for distributed systems and there is the problem of extending these to probabilistic distributed systems. Finally, it may be possible to extend the implementation relation defined for the reactive case by allowing races but using symbolic values to represent the probabilities of inputs supplied by the environment.

References

1. Ben-Abdallah, H., Leue, S.: Syntactic detection of process divergence and non-local choice in Message Sequence Charts. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 259–274. Springer, Heidelberg (1997)
2. Bravetti, M., Aldini, A.: Discrete time generative-reactive probabilistic processes with different advancing speeds. Theoretical Computer Science 290(1), 355–406 (2003)
3. Brinksma, E., Heerink, L., Tretmans, J.: Factorized test generation for multi-input/output transition systems. In: 11th IFIP Workshop on Testing of Communicating Systems, IWTCS 1998, pp. 67–82. Kluwer Academic Publishers, Dordrecht (1998)
4. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
5. Cheung, L., Stoelinga, M., Vaandrager, F.: A testing scenario for probabilistic processes. Journal of the ACM 54(6), Article 29 (2007)
6. Christoff, I.: Testing equivalences and fully abstract models for probabilistic processes. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 126–140. Springer, Heidelberg (1990)
7. Cleaveland, R., Dayar, Z., Smolka, S.A., Yuen, S.: Testing preorders for probabilistic processes. Information and Computation 154(2), 93–148 (1999)
8. Deng, Y., van Glabbeek, R., Hennessy, M., Morgan, C.: Characterising testing preorders for finite probabilistic processes. Logical Methods in Computer Science 4(4) (2008)
9. Frantzen, L., Merayo, M.G., Núñez, M.: A brief history of A-MOST. Journal of Logic and Algebraic Programming 78(6), 417–424 (2009)
10. van Glabbeek, R., Smolka, S.A., Steffen, B.: Reactive, generative and stratified models of probabilistic processes. Information and Computation 121(1), 59–80 (1995)

11. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Luettgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal methods to support testing. ACM Computing Surveys 41(2) (2009)
12. Hierons, R.M., Merayo, M.G.: Mutation testing from probabilistic and stochastic finite state machines. Journal of Systems and Software 82(11), 1804–1818 (2009)
13. Hierons, R.M., Merayo, M.G., Núñez, M.: Controllable test cases for the distributed test architecture. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 201–215. Springer, Heidelberg (2008)
14. Hierons, R.M., Merayo, M.G., Núñez, M.: Implementation relations for the distributed test architecture. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 200–215. Springer, Heidelberg (2008)
15. Hierons, R.M., Merayo, M.G., Núñez, M.: Implementation relations and test generation for systems with distributed interfaces (submitted, 2010)
16. Hierons, R.M., Ural, H.: The effect of the distributed test architecture on the power of testing. The Computer Journal 51(4), 497–510 (2008)
17. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
18. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE 84(8), 1090–1123 (1996)
19. López, N., Núñez, M., Rodríguez, I.: Specification, testing and implementation relations for symbolic-probabilistic systems. Theoretical Computer Science 353(1-3), 228–248 (2006)
20. Luo, G., Dssouli, R., von Bochmann, G.: Generating synchronizable test sequences based on finite state machine with distributed ports. In: 6th IFIP Workshop on Protocol Test Systems, IWPTS 1993, pp. 139–153. North-Holland, Amsterdam (1993)
21. Núñez, M.: Algebraic theory of probabilistic processes. Journal of Logic and Algebraic Programming 56(1-2), 117–177 (2003)
22. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)
23. Rafiq, O., Cacciari, L.: Coordination algorithm for distributed testing. The Journal of Supercomputing 24(2), 203–211 (2003)
24. Sarikaya, B., von Bochmann, G.: Synchronization and specification issues in protocol testing. IEEE Transactions on Communications 32, 389–395 (1984)
25. Segala, R.: Testing probabilistic automata. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 299–314. Springer, Heidelberg (1996)
26. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST 2008. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
27. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, San Francisco (2007)
28. Wu, S.-H., Smolka, S.A., Stark, E.W.: Composition and behaviors of probabilistic I/O automata. Theoretical Computer Science 176(1-2), 1–37 (1997)
29. Yi, W., Larsen, K.G.: Testing probabilistic and nondeterministic processes. In: 12th IFIP/WG6.1 Int. Symposium on Protocol Specification, Testing and Verification, PSTV 1992, pp. 47–61. North Holland, Amsterdam (1992)

Specification and Testing of E-Commerce Agents Described by Using UIOLTSs*

Juan José Pardo[1], Manuel Núñez[2], and M. Carmen Ruiz[1]

[1] Departmento de Sistemas Informáticos
Universidad de Castilla-La Mancha, Spain
juanjose.pardo@uclm.es, MCarmen.Ruiz@uclm.es
[2] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
mn@sip.ucm.es

Abstract. In this paper we expand our work in our specification formalism UIOLTSs. We present three implementation relations and provide alternative characterizations of these relations in terms of the tests that the implementation under test successfully passes. In addition, we present the main ideas to obtain an algorithm to derive complete test suites from specifications.

1 Introduction

During the software development process, it is very usual to apply structured methodologies, consisting of several phases such as analysis, specification, design, coding, and testing. Formal methods are a powerful tool that should be used along all the software development phases because they facilitate the descrition, analysis, validation and verification of software system. So developer can discover posible errors at the beginning of the development process.

Although it is very important to use formal methods to specify the behavior of the system, it is even more important to ensure that the implementation of the system is correct. In this line, *testing* is one of the most extended techniques to critically evaluate the quality of systems. Although testing and formal methods are considered rival, they are complimentary techniques that can profit from each other. The idea is that we have a formal model of the system (a specification), we check the correctness of the system under test by applying experiments and we match the results of these experiments with what the specification says and decide whether we have found an error. The formal description of the system allows to automatize most of the testing phases.

The main theory underlying formal specification and testing can be also applied to a specific kind of software like e-commerce agents. In this context, it is

* This research was partially supported by the TESIS project (TIN2009-14312-C02), by the Junta de Castilla-la Mancha project "Aplicación de métodos formales al diseño y análisis de procesos de negocio" (PEII09-0232-7745), and by the UCM-BSCH programme to fund research groups (GR58/08 - group number 910606).

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 78–86, 2010.
© IFIP International Federation for Information Processing 2010

necessary to introduce new features in the formal language in order to express the high-level requirements of agents, which are usually defined in economic terms. In the literature, we can find several proposals to use formal methods to formalize multi-agent systems (see [1]).

The initial point of this paper can be found in one formalism previously developed within our research group [2]. In that paper we presented a formalism called *utility state machines*, which was based on finite state machines with a strict alternation between inputs and outputs and, where the user's preferences are defined by means of *utility functions* associating a numerical value to each possible set of resources that the system can trade. The alternation between inputs and outputs is a ver strong restriction that we wanted avoid in our model but, this slightly complicates the semantic framework. In particular, we need to include the notion of *quiescence* to characterize states that cannot produce outputs and we have to redefine the notion of test and how to apply tests to systems. On the contrary, we have reduced some of the complexity associated with our previous formalism.

Our new formalism, called *Utility Input-Output Labeled Transition System* (in short, UIOLTS), was presented in [3]. In this paper we complete our previous work by defining three implementation relations that can be used to formally establish the conformance of a system under test with respect to a specification. One of them takes into account only the sequences of inputs and outputs produced by the system and the other two relations consider resources that the system has after an action is executed. We redefine the notion of test so that we can obtain more information from the system under test. In order to relate the application of tests and our implementation relations, we define an algorithm to derive complete test suites from specifications, that is, an implementation conforms to a specification if and only if successfully passes the test suite produced from the specification.

The rest of the paper is structured as follows. In Section 2 we introduce our formalism. In Section 3 we define our implementation relations. In Section 4 we give the notion of test and how to apply tests to implementations under test. Finally, in Section 5 se present our conclusions and some lines for future work.

2 A Framework to Formally Specify Economic Agents

In this section we present our formalism as defined in [3]. Basically, a UIOLTS is a labeled transition system where we introduce some new features to define agent behaviors in an appropriate way. The first new element that we add is a set of variables, where each variable represents the amount of the resource that the system owns. In addition, we associate a utility function to each state of the system. This utility function can be used to decide whether the agent accepts an exchange of resources proposed by another agent. Intuitively, given a utility function u we have that $u(\overline{x}) < u(\overline{y})$ means that the basket of resources represented by $\overline{y}$ is preferred to $\overline{x}$.

Definition 1. We consider $\mathbb{R}_+ = \{x \in \mathbb{R} \,|\, x \geq 0\}$. We will usually denote *vectors* in $\mathbb{R}^n$ (for $n \geq 2$) by $\overline{x}, \overline{y}, \overline{v} \ldots$ Given $\overline{x} \in \mathbb{R}^n$, x_i denotes its i-th component. We extend to vectors some usual arithmetic operations. Let $\overline{x}, \overline{y} \in \mathbb{R}^n$. We define the addition of vectors $\overline{x}$ and $\overline{y}$, denoted by $\overline{x}+\overline{y}$, simply as $(x_1+y_1, \ldots, x_n+y_n)$. We write $\overline{x} \leq \overline{y}$ if for all $1 \leq i \leq n$ we have $x_i \leq y_i$.

We will suppose that there exist $n > 0$ different kinds of resources. *Baskets of resources* are defined as vectors $\overline{x} \in \mathbb{R}^n_+$. Therefore, $x_i = r$ denotes that we own r units of the i-th resource. A *utility function* is a function $u : \mathbb{R}^n_+ \longrightarrow \mathbb{R}$. In microeconomic theory there are some restrictions that are usually imposed on utility functions (mainly, strict monotonicity, convexity, and continuity). □

Our systems can perform two different types of actions. *Output actions* are initiated by the system and cannot be refused by the environment. We consider that the performance of an output action can cost resources to the system. In addition, the performance of an output action will usually have an associated condition to decide whether the system performs it or not. *Input actions* are initiated by the environment and cannot be refused by the system, that is, we consider that our systems under test are *input-enabled* (specifications do not need to be input-enabled). The performance of an input action can increase the resources of the agent that performs it. In addition to these two types of actions we need a third type that we introduce for technical reasons to represent quiescence [4]. This special action is denoted by δ, and special transitions labeled by this same δ action. In the following definition we also introduce the notion of *configuration*. Usually, in order to clearly identify *where* a system is, it is enough to record the current state. In our setting, in order to record the current situation of an agent we use pairs where we keep the current state of the system and the current amount of available resources.

Definition 2. A *Utility Input Output Labeled Transition System*, in short UIOLTS, is a tuple $M = (S, s_0, L, T, U, V)$ where

- S is the set of states, being $s_o \in S$ the initial state.
- V is an n-tuple of resources belonging to R_+. We denote by $\overline{v}_0$ the initial tuple of values associated with these resources.
- L is the set of actions. The set of actions is partitioned into three pairwise disjoint sets: the set of imputs actions L_I which elements are preceded by ?, the set of output actions L_O which elements are preceded by ! and a set with one special ation δ that represents quiescence.
- T is the set of transitions that is partitioned into three pairwise disjoint sets: the set of input transitions T_I which elements are tuples $(s, ?i, \bar{x}, s_1)$ where $\bar{x} \in \mathbb{R}^n_+$ is the increase in the set of resources, the set of output transitions T_O which elements are tuples $(s, !o, \bar{z}, C, s_1)$ where $\bar{z} \in \mathbb{R}^n_+$ is the decrease in the set of resources, and C is a predicate on the set of resources and the set of quiescence transtions with tuples $(s, \delta, \overline{0}, C_s, s)$ where $C_s = \bigwedge_{(s,!o,\bar{z},C,s_1) \in T_O} \neg C$.
- $U : S \rightarrow (\mathbb{R}^n_+ \longrightarrow R_+)$ is a function associating a utility function to each state in S.

A *configuration* of M is a pair $(s, \bar{v})$, where s is the current state and $\bar{v}$ is the current value of V. We denote by $Conf(M)$ the set of configurations of M.

We say that M is *input-enabled* if for all $s \in S$ and $?i \in L_I$ there exist $\bar{x}$ and s_1 such that $(s, ?i, \bar{x}, s_1) \in T_I$. □

Now we can define the concatenation of several transitions of an agent to capture the different evolutions, from one configuration to another one, that an agent can carry out. These evolutions can be produced either by executing an input or an output action or by offering an *exchange of resources*. As we will see, exchanges of resources have low priority and will be allowed only if no output can be performed. The idea is that if we can perform an output with the existing resources, then we do not need to exchange resources.

Definition 3. Let $M = (S, s_0, L, T, U, V)$ be a UIOLTS. We consider that M can evolve from the configuration $c = (s, \overline{v})$ to the configuration $c' = (s', \overline{v'})$ if one of the following options is possible:

1. If there is an input transition $(s, ?i, \bar{x}, s_1)$, then this transition can be executed. The new configuration is $c' = (s_1, \overline{v} + \bar{x})$.
2. If there is an output transition $(s, !o, C, \bar{z}, s_1)$ such that $C(\overline{v})$ holds then the transition can be executed. The new configuration is $c' = (s_1, \overline{v} - \bar{z})$.
3. Let us consider the transition associated with quiescence at s: $(s, \delta, C_s, \bar{0}, s)$. If $C_s(\overline{v})$ holds, that is, no output transition is currently available, then this transition can be executed. The configuration is not altered, that is, $c' = (s, \overline{v})$.
4. Let us consider again the transition associated with quiescence at s, that is, $(s, \delta, C_s, \bar{0}, s)$. If $C_s(\overline{v})$ holds, then we can offer an exchange. We represent an exchange by a pair $(\xi, \bar{y})$ where $\bar{y} = (y_1, y_2, \dots y_n) \in \mathbb{R}^n$ is the variation of the set of resources. Therefore, $y_i < 0$ indicates a decrease of the resource i while $y_i > 0$ represents an increase of the resource i. M will be willing to perform an exchange $(\xi, \bar{x})$ if $U(s, \overline{v}) < U(s, \overline{v} + \bar{x})$. If another agent is accepting the exchange, then the new configuration is $c' = (s, \overline{v} + \bar{y})$.

We denote an evolution from the configuration c to the configuration c' by the triple $(c, (a, \bar{y}), c')$, where $a \in L \cup \{\xi\}$ and $\bar{y} \in \mathbb{R}^n$. We denote by $Evolutions(M, c)$ the set of evolutions of M from the configuration c and by $Evolutions(M)$ the set of evolutions of M from (s_0, v_0), the initial configuration.

A trace of M is a finite sequence of evolutions. $Traces(M, c)$ denotes the set of traces of M from the configuration c and $Traces(M)$ denotes the set of traces of M from the initial configuration. Let $l = e_1, e_2, \dots, e_m$ be a trace of M where for all $1 \leq i \leq m$ we have $e_i = (c_i, (a_i, \bar{x_i}), c_{i+1})$. The observable trace associated to l is a triple (c_1, σ, c_{n+1}), where σ is the sequence of actions obtained from $a_1, a_2, \dots, a_m$ by removing all occurrences of ξ. We sometimes represent this observable trace as $c_1 \overset{\sigma}{\Longrightarrow} c_{n+1}$. □

3 Implementation Relations for UIOLTSs

In this section we introduce our implementation relations to formally establish when an implementation is correct with respect to a specification. In our context,

the notion of correctness has several possible definitions. For example, a user of our methodology may consider that an implementation I of a specification S is good if the number of resources that I obtains after performing some actions is always greater than the one given by S while another user could be happy with an agent that obtains smaller amounts of resources as long as the utility returned by them is bigger than the one foreseen by the specification. Let us remind that implementations must be input-enabled while specifications might not be.

We have defined three different implementation relations. The first one is close to the classical **ioco** implementation relation [5] where an implementation I is correct with respect to a specification S if the output actions executed by I after a sequence of actions is performed are a subset of the ones that can be executed by S. Intuitively, this means that the implementation does not *invent* actions that the specification did not contemplate. The formal definition of our first implementation relation was presented in [3].

In order tu define our two new implementation relation we introduce some auxiliary notation.

Definition 4. Let $M = (S, s_0, L, T, U, V)$ be a UIOLTS, $c = (s, \overline{x}) \in Conf(M)$ a configuration of M, and $\sigma \in L^*$ be a sequence of actions. Then,

$$c \texttt{ after } \sigma = \{c' \in Conf(M) | c \stackrel{\sigma}{\Longrightarrow} c'\}$$

We use M after σ as a shorthand for c_0 after σ, being c_0 the initial configuration of M. □

Intuitively, c after σ returns the configuration reached from the configuration c by the execution of the trace σ.

Our firts new implementation relation is based on the **ioco** mechanism but we take into account both the resources that the system has and the actions that the system can execute. In order to define the new relation we need to define the set out of outputs. In this case we have two components: The output action that can be executed and the set of resources that the system has. We also introduce an operator to compare sets of pairs (output,resources).

Definition 5. Let $M = (S, s_0, L, T, U, V)$ be a UIOLTS and $c = (s, \bar{x}) \in Conf(M)$ be a configuration of M. Then,

$$\begin{aligned} \texttt{out}'(c) = \{(!o, \bar{y}) \in L_O \times \mathbb{R}^n_+ | \exists s_1, \bar{z}, C : (s, !o, C, \bar{z}, s_1) \in T \wedge C(\overline{x}) \wedge \bar{y} = \bar{x} - \bar{z}\} \\ \cup \{(\delta, \bar{x}) | \exists C_s : (s, \delta, C_s, \bar{0}, s) \in T \wedge C_s(\overline{x})\} \end{aligned}$$

We extend this function to deal with sets of configurations in the expected way, that is, $\texttt{out}'(C) = \bigcup_{c \in C} \texttt{out}'(c)$.

Given two sets $A = \{(o_1, \bar{y_1}), \ldots, (o_n, \bar{y_n})\}$ and $B = \{(o_1, \bar{x_1}), \ldots, (o_n, \bar{x_n})\}$, we write $A \sqsubseteq B$ if $act(A) \subseteq act(B)$ and for all output action $!o \in act(A)$ we have $\min(rec(A, o)) \geq max(rec(B, o))$, where $Act(X) = \{a | (a, \bar{y}) \in X\}$ and $rec(X, o) = \{r | (o, r) \in X\}$. □

The set $\texttt{out}'(c)$ contains those actions (outputs or quiescence) that can be performed when the system is in configuration c as well as the set of resources obtained after their performance. Next, we introduce our new implementation relation. We consider that an implementation I is correct with respect to a specification S if the output actions performed by the implementation in a state are a subset of those that can be performed by the specification in this state and the set of resources of implementation I is *better* than the set of resources in the specification.

Definition 6. Let I, S be two UIOLTSs with the same set of actions L. We write I $\mathbf{ioco_r}$ S if for all sequence of actions $\sigma \in \mathit{Traces}(S)$ we have that $\texttt{out}'(I \texttt{ after } \sigma) \sqsubseteq \texttt{out}'(S \texttt{ after } \sigma)$. □

Our new second implementation relation is again based on the **ioco** approach but we take into account both the utility value that the available resources provide and the actions that the system can execute. In order to define the new relation we need to redefine the set of immediately available outputs. In this case, our set of outputs has two components: The output action that can be executed and the value of the utility function after this action is executed. We also introduce an operator to compare sets of pairs (output,utility).

Definition 7. Let $M = (S, s_0, L, T, U, V)$ be a UIOLTS and $c = (s, \bar{x}) \in \mathit{Conf}(M)$ be a configuration of M. Then,

$$\begin{aligned}\texttt{out}''(c) = \{(!o, U(s_1, \bar{x} - \bar{z})) \in L_O \times \mathbb{R}_+ | \exists s_1, \bar{z}, C : (s, !o, C, \bar{z}, s_1) \in T \wedge C(\overline{x})\} \\ \cup \{(\delta, U(s, \bar{x})) | \exists C_s : (s, \delta, C_s, \bar{0}, s) \in T \wedge C_s(\overline{x})\}\end{aligned}$$

We extend this function to deal with sets of configurations in the expected way, that is, $\texttt{out}''(C) = \bigcup_{c \in C} \texttt{out}''(c)$.

Given two sets $A = \{(o_1, u_1), \ldots, (o_n, u_n)\}$ and $B = \{(o_1, u_1), \ldots, (o_n, u_n)\}$, we write $A \sqsubseteq' B$ if $\mathit{act}(A) \subseteq \mathit{act}(B)$ and for all output action $!o \in \mathit{act}(A)$ we have $\min(\mathit{util}(A, o)) \geq \mathit{max}(\mathit{util}(B, o))$, where $\mathit{act}(X) = \{a | (a, y) \in X\}$ and $\mathit{util}(X, o) = \{u | (o, u) \in X\}$. □

The set $\texttt{out}''(c)$ contains those actions (outputs or quiescence) that can be performed when the system is in configuration c as well as the value of the utility function obtained after their performance. Next, we introduce our new implementation relation. We consider that an implementation I is correct with respect to a specification S if the output actions performed by the implementation in a state are a subset of those that can be performed by the specification in this state and the value of the utility function of implementation I is *better* than the value of the utility function in the specification.

Definition 8. Let I, S be two UIOLTSs with the same set of actions L. We write I $\mathbf{ioco_u}$ S if for all sequence of actions $\sigma \in \mathit{Traces}(S)$ we have that $\texttt{out}''(I \texttt{ after } \sigma) \sqsubseteq' \texttt{out}''(S \texttt{ after } \sigma)$. □

4 Tests: Definition and Application

A *test* represents an experiment that will be carried out on an implementation under test (IUT). Depending on the answers provided by the IUT we may conclude that it is behaving in an unexpected way. In our setting, a test can do three different things: It can accept an output action started by the implementation, it can provide an input action to the implementation, or it can propose a exchange of resources. If the test receives an output, then it checks whether the action belongs to the set of expected ones (according to its description); if the action does not belong to this set, then the tester will produce a fail signal. In addition, each state of a test saves information about the set of resources that the tested system has if the test reaches this state. Therefore, we might also detect errors if the amounts of resources differ from the ones that the test indicates.

In our framework, a test for a system is modeled by a UIOLTS, where its set of input actions is the set of output actions of the specification and its set of output actions is the set of input actions of the specification. Also, we include a new action θ that represents the observation of quiescence. In order to be able to accept any output from the tested agent, we consider that tests are *input-enabled*, since its inputs correspond to outputs of the tested agent. Let us remark that the current notion of test is more involved that the one given in [3] since the latter did not include any mechanism to deal with the amount of resources available to the implementation under test.

Definition 9. Let $M = (S, s_0, L, T, U, V)$ be and UIOLTS, with $L = L_I \cup L_O \cup \{\delta\}$. A test for M is a UIOLTS $t = (S^t, s_0^t, L^t, T^t, \lambda, V)$ where

- S^t is the set of states, where $s_0^t \in S^t$ is the initial state and there are two special states called `fail` and `pass`, with `fail` $\neq$ `pass`.
- $\lambda : S \rightarrow \mathbb{R}_+^n$ is a function that assigns a tuple of real numbers to a state. This tuple represents the amount of each resource in this state.
- L^t is the set of actions where L_I is the set of outputs of M, L_O is the set of imputs of M, θ is a special action that represents the detection of quiescence and ξ is an special action that represents the proposal of an exchange.
- $T^t = T_e \cup T_\theta$ is the set of transitions, where
  - $T_e \subseteq S^t \times L_O \cup L_I \cup \{\xi\} \times \mathbb{R}^n \times S^t$ is the set of *regular* transitions.
  - $T_\theta \subseteq S^t \times \{\delta\} \times S^t$. □

Our tests can compare the resources that the IUT has and the ones properly specified in the test. Therefore, it is suitable to *test* systems according to the ideas underlying the first two implementation relations. If we are interested only in the returned utility (regardless of the specific amounts of different resources), we have to replace the definition of λ by the following: $\lambda : S \rightarrow (\mathbb{R}_+^n \rightarrow \mathbb{R})$ is a function that assigns a utility function to each state in S.

We define configurations of a test in the same way that we used to define them for UIOLTSs, and we thus omit the definition.

Given an implementation I and a test t, running t with I is the synchronized parallel execution of both taking into account the peculiarities of the special actions δ, θ, and ξ.

A first notion of passing a test considers only that the actions that the IUT performs are the expected ones.

Definition 10. A test execution of the test t with an implementation I is a trace of $I||\lceil t$ leading to one of the states `pass` or `fail` of t.

We say that an implementation I passes a test t if all test executions of t with I go to a `pass` state of t. □

Another more complex notion for passing a test, considering the resources administered by the system, is the following.

Definition 11. An implementation I passes$_r$ a test t if all test execution σ of t with I reaches a `pass` state s of t and $rec(I \texttt{ after } \sigma) \geq rec(S \texttt{ after } \sigma)$. □

The previous definition can be modified to deal with the alternative notion of test discussed at the end of Definition 9 where we do not compare resources but only consider the utility returned by the available resources.

Definition 12. An implementation I passes$_u$ a test t if all test execution σ of t with I reaches a `pass` state s of t and $util(I \texttt{ after } \sigma) \geq util(S \texttt{ after } \sigma)$. □

These three notions can be easily extended to deal with set of tests in the expected way: If $\mathcal{T}$ is a test suite then we say that I passes$_x$ $\mathcal{T}$ if for all $t \in \mathcal{T}$ we have I passes$_x$ t.

After definition of test we need to define an algorithm to derive test from specifications. Due to space limitation we do not show the algorithm.

Our algorithm is non-deterministic in the sense that there exist situations where different possibilities are available, and we have different tests depending on the choice that we select. If we consider all the possible choices we will have a full test suite. We denote the test suite produced by the algorithm for a specification M by $Test(M)$. Now we can present results that relate, for a specification S and an implementation I, the application of test suites derived from the specification and the different implementation relations. we omit the proof of this theorem due to space-limitations.

Theorem 1. *Lets S, I be UIOLTSs. We have I* **ioco**$_*$ *S if and only if I passes$_*$ $tests(S)$, where $*$ is r or u or nothing .*

5 Conclusions and Future Work

We have recently defined a new formalism, called *Utility Input Output Labeled Transition Systems*, to specify the behavior of e-commerce agents. In this paper we have introduced a testing methodology, based on this formalism, to test whether an implementation of a specified agent behaves as the specification says that it behaves. We have defined three different implementation relations, a notion of test, and an algorithm to obtain, from a given specification, a set of *relevant* tests.

Concerning future work, we currently focus on two research lines. The first one is based on theoretical aspects and we would like to extend our formalism in order to specify the behavior of agents that are influenced by the passing of time and would like to define the interaction between agents in order to test multi-agents systems. The second line is more practical since we would like to apply our formalism to real complex agents. In order to support this line of work, we are developing a tool to automatically generate tests from specifications and apply them to implementations.

References

1. Núñez, M., Rodríguez, I., Rubio, F.: Formal specification of multi-agent e-barter systems. Science of Computer Programming 57(2), 187–216 (2005)
2. Núñez, M., Rodríguez, I., Rubio, F.: Specification and testing of autonomous agents in e-commerce systems. Software Testing, Verification and Reliability 15(4), 211–233 (2005)
3. Pardo, J.J., Núñez, M., Ruiz, M.C.: A novel formalism to represent collective intelligence in multi-agent systems. In: New Challenges in Computational Collective Intelligence. SCI, vol. 244, pp. 193–204. Springer, Heidelberg (2009)
4. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software – Concepts and Tools 17(3), 103–120 (1996)
5. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST 2008. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)

Testing Attribute-Based Transactions in SOC

Laura Bocchi and Emilio Tuosto

Department of Computer Science, University of Leicester, UK

Abstract. We set the basis for a theory of testing for distributed transactions in service oriented systems where each service definition is decorated with a *transactional attribute* (inspired by the Java Transaction API). Transaction attributes discipline how services are executed with respect to the transactional scope of the invoking party.

We define a language of observers and show that, in general, the choice of different transactional attributes causes different system's behaviours wrt the testing equivalences induced by the observers.

1 Introduction

We give an observational theory for transactional behaviours in *Service-Oriented Computing* (SOC) based on the theory of testing [4]. Transaction in SOC, often referred to as *long-running*, feature a mechanism called *compensation* which is a weaker version[1] of the classic rollback mechanism of ACID transactions in database systems. In SOC, each activity of a transactional computation can be associated with a compensation *installed* as the activity is executed. The run-time failure of an activity is backwardly propagated and triggers the execution of the compensations installed for the activities completed earlier. Therefore, compensations have been studied in relation to mechanisms of failure propagation.

Notably, the key characteristics of SOC are loose-coupling and dynamism: services can be discovered at run-time relying only on their published interface, and upon service invocation the system dynamically reconfigures to include the newly created service instance. System reconfigurations should also consider transactional scopes (or scopes for short) as they play a fundamental role in failures propagation.

Consider the system $\langle \mathtt{invoke}(s).P \rangle$ where the transaction, represented by the angled brackets, includes a process that invokes a service, which is described by the interface s, and then behaves like P. Suppose that there exists a provider that implements s as process Q. Should the system evolve so to include Q in the scope of the invoking process (i.e., $\langle P \mid Q \rangle$)? Should Q be running in a fresh scope (i.e., $\langle P \rangle \mid \langle Q \rangle$)? Or else, should Q be outside any scope (i.e., $\langle P \rangle \mid Q$)? Each alternative is valid and influences failure propagation and the behaviour of the system (as shown in § 4).

We design an observational theory that yields a formal framework for analysing the interplay between communication failures and the behaviour of a service-oriented system. We use *may*- and *must*-testing equivalences to compare transactional behaviours.

[1] ACID transactions are implemented by locking resources. Locks can be unfeasible if transactions are long lasting.

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 87–94, 2010.
© IFIP International Federation for Information Processing 2010

Table 1. Informal semantics of EJB attributes. Boxes represent scopes, $\bullet$ represent callers, $\circ$ represent callees. Failed activities are denoted by $\otimes$. Each row shows the behaviour of one attribute; the first two columns show, respectively, invocations from outside and from within a scope.

	invoker **outside** a scope	invoker **inside** a scope	callee supports
(1)	$\bullet \Longrightarrow \bullet\boxed{\circ}$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet\ \circ}$	`r` (**Requires**)
(2)	$\bullet \Longrightarrow \bullet\boxed{\circ}$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet}\boxed{\circ}$	`rn` (**Requires New**)
(3)	$\bullet \Longrightarrow \bullet\ \circ$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet}\ \circ$	`ns` (**Not Supported**)
(4)	$\bullet \Longrightarrow \otimes$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet\ \circ}$	`m` (**Mandatory**)
(5)	$\bullet \Longrightarrow \bullet\ \circ$	$\boxed{\bullet} \Longrightarrow \boxed{\otimes}$	`n` (**Never**)
(6)	$\bullet \Longrightarrow \bullet\ \circ$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet\ \circ}$	`s` (**Supported**)

A remarkable feature of our framework is that it allows to discipline the reconfiguration of transactional scopes, hence to predict and control the effects of failures in the reconfigured system.

We build up on ATc (after *Attribute-based Transactional calculus*) [1], a CCS-like process calculus designed to model dynamic SOC transactions featuring EJB transactional attributes [7,6]; ATc and EJB attributes are summarised in § 2. § 3 yields the main contribution of the paper, namely the definition of a class of observers which induces suitable testing equivalences to compare ATc systems as shown in § 4.

2 Background

The ATc calculus presented in [1] takes inspiration from the *Container Managed Transactions* (CMT) mechanism of Enterprise Java Beans (EJB). Hereafter, the terms *container* and *service provider* which refer to the environment where methods and services are executed, will be used interchangeably.

An ATc *container* associates each service interface to a *transactional attribute* (attribute, for short) which specifies (*i*) the 'reaction' of the system upon invocations (e.g., "*calling the service from outside a scope throws an exception*"), and (*ii*) how scopes dynamically reconfigure (e.g., "*the invoked service is always executed in a newly created scope*"). On the other hand, also the invoking party can specify which attribute must be supported by the invoked service. This is natural in SOC where, typically, the service properties are mutually negotiated between requester and provider.

The set of attributes is

$$\mathcal{A} \stackrel{def}{=} \{\mathtt{m}, \mathtt{s}, \mathtt{n}, \mathtt{ns}, \mathtt{r}, \mathtt{rn}\} \qquad (\textit{attributes})$$

The intuitive semantics of each $a \in \mathcal{A}$ (attributes range over $a, a_1, a_2, \ldots$) is in Table 1.

An ATc process is a CCS-like process with three additional capabilities: *service invocations*, *transactional scoping*, and *compensation installation*. The set $\mathcal{P}$ of ATc *processes* is given by the following grammar

$$P, Q ::= \mathtt{0} \mid \nu x\, P \mid P \mid Q \mid !P \mid s \Leftarrow A.P \mid \langle P \rangle_Q \mid \pi \downarrow Q.P \mid \mathtt{err} \qquad (processes)$$

where $s, s', \ldots$ range over a set of *service* names $\mathcal{S}$ while $x, y, z, \ldots$ range over a *channel* names $\mathcal{N}$ (assumed to be both countably infinite and disjoint), u ranges over $\mathcal{S} \cup \mathcal{N}$, and π is either x or $\overline{x}$. We assume $x = \overline{\overline{x}}$. Restriction $\nu x\, P$ binds x in P; we denote the sets of *free* and *bound* channels of $P \in \mathcal{P}$ by $\mathsf{fc}(P)$ and $\mathsf{bc}(P)$. The standard process algebraic syntax is adopted for idle process, restriction, parallel composition, and replication. Process $s \Leftarrow A.P$ invokes a service s required to support one of the attributes in $A \subseteq \mathcal{A}$; a scope $\langle P \rangle_Q$ consists of a running process P and a compensation Q (confined in the scope) to be executed only upon failure (scopes can be nested); $\pi \downarrow Q.P$ executes π and installs the compensation Q in the enclosing scope (if any), then behaves as P; finally, $\mathtt{err}$ represents a run-time failure ($\mathtt{err}$ cannot be used by programmers).

A *system*

$$\Gamma \vdash P \quad \text{with } \Gamma = \{\gamma_1, \ldots, \gamma_n\} \quad (systems) \qquad \gamma : \mathcal{S} \to \mathcal{A} \times \mathcal{P} \quad (containers)$$

is a process P within an *environment* Γ, namely within a set of *containers*. A container is a finite partial map that assign an attribute and a "body" to service names. When defined, $\gamma(s) = (a, P)$ ensures that, if invoked in γ, s supports the attribute a and activates an end-point that executes as P. Environments may offer different implementations of s and support different attributes. Henceforth we write $P \in \Gamma(s, A)$ for $\exists \gamma \in \Gamma\ \exists a \in A\ :\ \gamma(s) = (a, P)$ and $P \in \Gamma(s, a)$ for $P \in \Gamma(s, \{a\})$.

The semantics of communications is given in terms of *contexts*; $\mathbf{C}[\boxdot]$ is *scope-avoiding* (s-a, for short) if there are no $P, Q \in \mathcal{P}$ and $\mathbf{C}'[\boxdot]$ s.t. $\mathbf{C}[\boxdot] = \mathbf{C}'[\langle \boxdot \mid P \rangle_Q]$.

$$\mathbf{C}[\boxdot] ::= \boxdot \;\Big|\; \langle \mathbf{C}[\boxdot] \mid P \rangle_Q \;\Big|\; P \mid \mathbf{C}[\boxdot] \;\Big|\; \mathbf{C}[\boxdot] \mid P \qquad (contexts)$$

The *reduction relation of ATc processes* (i.e., $\to$) is the smallest relation $\to \subseteq \mathcal{P} \times \mathcal{P}$ closed under the following axioms and rules:

$$\mathbf{C}[\langle \pi \downarrow Q.P \rangle_R] \mid \mathbf{C}'[\langle \bar{\pi} \downarrow Q'.P' \rangle_{R'}] \;\to\; \mathbf{C}[\langle P \rangle_{R|Q}] \mid \mathbf{C}'[\langle P' \rangle_{R'|Q'}] \qquad \texttt{(p1)}$$

$$\mathbf{C}[\langle \pi \downarrow Q.P \rangle_R] \mid \mathbf{C}'[\bar{\pi} \downarrow Q'.P'] \;\to\; \mathbf{C}[\langle P \rangle_{R|Q}] \mid \mathbf{C}'[P'], \quad \text{if } \mathbf{C}'[\boxdot] \text{ is s-a} \qquad \texttt{(p2)}$$

$$\mathbf{C}[\pi \downarrow Q.P] \mid \mathbf{C}'[\bar{\pi} \downarrow Q'.P'] \;\to\; \mathbf{C}[P] \mid \mathbf{C}'[P'], \quad \text{if } \mathbf{C}[\boxdot] \text{ and } \mathbf{C}'[\boxdot] \text{ are s-a} \qquad \texttt{(p3)}$$

$$\frac{P \to P'}{P \mid R \to P' \mid R} \qquad \frac{P \to P'}{\nu x\, P \to \nu x\, P'} \qquad \frac{P \equiv P' \to Q' \equiv Q}{P \to Q} \qquad \texttt{(p4÷ p6)}$$

The $\to$ relation is defined up-to a standard structural congruence relation $\equiv$ (which is extended to contexts). In `(p1÷ p3)`, sender and receiver synchronise regardless the relative nesting of their scopes. Upon synchronisation, compensations are installed in parallel to the other compensations of the enclosing scope; if $\mathbf{C}[\boxdot]$ is s.a. then compensations are discarded.

The *reduction relation of ATc systems* (i.e., $\rightsquigarrow$) is defined below, assuming $\mathbf{C}[\square] \neq \mathsf{0}$.

$$\frac{P \to P'}{\Gamma \vdash P \rightsquigarrow \Gamma \vdash P'} \qquad \frac{\mathtt{m} \in A \qquad \mathbf{C}[\square] \text{ is s-a}}{\Gamma \vdash \mathbf{C}[s \mathrel{\varepsilon\!\!-} A.P] \rightsquigarrow \Gamma \vdash \mathbf{C}[\mathtt{err}]} \tag{s1/s2}$$

$$\frac{R \in \Gamma(s, \{\mathtt{s}, \mathtt{n}, \mathtt{ns}\} \cap A) \qquad \mathbf{C}[\square] \text{ is s-a}}{\Gamma \vdash \mathbf{C}[s \mathrel{\varepsilon\!\!-} A.P] \rightsquigarrow \Gamma \vdash \mathbf{C}[P] \mid R} \quad \frac{R \in \Gamma(s, \{\mathtt{r}, \mathtt{rn}\} \cap A) \qquad \mathbf{C}[\square] \text{ is s-a}}{\Gamma \vdash \mathbf{C}[s \mathrel{\varepsilon\!\!-} A.P] \rightsquigarrow \Gamma \vdash \mathbf{C}[P] \mid \langle R \rangle} \tag{s3/s4}$$

$$\frac{P = \mathbf{C}[\langle s \mathrel{\varepsilon\!\!-} A.P_1 \mid P_2 \rangle_Q] \qquad \mathsf{bc}(P) \cap \mathsf{fc}(R) = \emptyset \quad R \in \Gamma(s, \{\mathtt{m}, \mathtt{s}, \mathtt{r}\} \cap A)}{\Gamma \vdash P \rightsquigarrow \Gamma \vdash \mathbf{C}[\langle P_1 \mid P_2 \mid R \rangle_Q]} \tag{s5}$$

$$\frac{\mathtt{n} \in A}{\Gamma \vdash \mathbf{C}[\langle s \mathrel{\varepsilon\!\!-} A.P_1 \mid P_2 \rangle_Q] \rightsquigarrow \Gamma \vdash \mathbf{C}[Q]} \tag{s6}$$

$$\frac{\mathtt{rn} \in A \wedge R \in \Gamma(s, \mathtt{rn})}{\Gamma \vdash \mathbf{C}[\langle s \mathrel{\varepsilon\!\!-} A.P_1 \mid P_2 \rangle_Q] \rightsquigarrow \Gamma \vdash \mathbf{C}[\langle P_1 \mid P_2 \rangle_Q] \mid \langle R \rangle} \tag{s7}$$

The rules above correspond to the informal presentation in Table 1: (s2÷ s4) model the first column and (s5÷ s7) model the second one. Failures trigger the compensation when occurring inside a scope (s6) and lead to an error otherwise (s2).

ATc systems do not model communication failures[2] and do not provide an explicit notion of commit for transactions. These aspects are modelled in § 3.

3 Observers for ATc

In this section we provide a theory of testing by defining a notion of *observers* suitable for ATcthat interact with systems and possibly cause communication failures. Two systems are equivalent if they cannot be distinguished by an observers (they "pass the same tests").

In § 3.1 we define observers and observed systems, in § 3.2. we give an observational semantics of ATc, in § 4 we show some motivating examples.

3.1 Observed Systems

The class of observers defined in this section is used to model communication failures and define successful computations. An *observer* is derived by the following grammar:

$$O ::= \mathsf{0} \;\big|\; \checkmark \;\big|\; \pi.O \;\big|\; \lightning\pi.O \;\big|\; O + O \;\big|\; \mathbf{rec}\ X.O \;\big|\; X \qquad (\textit{observers})$$

The *structural congruence for observers* is the smallest equivalence relation closed under the monoidal axioms of + and it is denoted as $\equiv_o$.

We consider sequential observers. Failing and successful tests are represented by $\mathsf{0}$ and $\checkmark$, respectively; prefix $\pi.O$ allows observers to communicate with the system, while prefix $\lightning\pi.O$ causes the failure of π in the system and continues as O; observers can be composed with the (external) choice operator + and recursively defined as $\mathbf{rec}\ X.O$ (where the occurrences of X in O are supposed guarded by prefixes). An observer is a process that can interact with a system over its (free) channels and trigger failures in the communications (e.g., to check that failures are correctly handled). Since observers

[2] The relation $\rightsquigarrow$ only considers errors due to misuse of attributes.

cannot be composed in parallel, they do not communicate among themselves. This, and the absence of name passing in ATc, allow us to avoid using name restriction in observers. Moreover, observers do not run in transactional scopes and they are not allowed to invoke services; they are used to model communication failures so to scrutinize the transactional behaviour of ATc systems.

Let systems be ranged over by $S, S', \ldots$; the set States of *observed systems* is the set of pairs made of a system S and an observer O, written as $S \parallel O$.

The *reduction relation of ATc observed systems* (i.e., $\rightsquigarrow$) is the smallest relation satisfying the following axioms (where $\mathbf{C}[\boxtimes]$ is s-a in (os1/os2)):

$$\Gamma \vdash \mathbf{C}[\pi \downarrow Q.P] \parallel \overline{\pi}.O \rightsquigarrow \Gamma \vdash \mathbf{C}[P] \parallel O \qquad \Gamma \vdash \mathbf{C}[\langle \pi \downarrow Q.P \rangle_R] \parallel \overline{\pi}.O \rightsquigarrow \Gamma \vdash \mathbf{C}[\langle P \rangle_{Q|R}] \parallel O \quad \texttt{(os1/os2)}$$

$$\Gamma \vdash \mathbf{C}[\pi.P] \parallel \lightning\pi.O \rightsquigarrow \Gamma \vdash \mathbf{C}[\mathtt{err}] \parallel O \qquad \Gamma \vdash \mathbf{C}[\langle \pi.P \mid R \rangle_Q] \parallel \lightning\pi.O \rightsquigarrow \Gamma \vdash \mathbf{C}[Q] \parallel O \quad \texttt{(os3/os4)}$$

$$S \parallel \checkmark \rightsquigarrow S \parallel \checkmark \qquad \frac{O \equiv_o O_1 \quad S \parallel O_1 \rightsquigarrow S' \parallel O_2 \quad O_2 \equiv_o O'}{S \parallel O \rightsquigarrow S' \parallel O'} \quad \texttt{(os5/os6)}$$

$$\frac{S \leadsto S'}{S \parallel O \rightsquigarrow S' \parallel O} \qquad \frac{S \parallel O \rightsquigarrow S' \parallel O'}{S \parallel O + O'' \rightsquigarrow S' \parallel O'} \quad \texttt{(os7/os8)}$$

Rules (os1/os2) model a communication step involving the system and the observer. Communication failures occurring outside a scope yield an error (os3); failures occurring inside a scope trigger the compensations associated with the enclosing scope (os4). Rule (os5) signals when a test is passed, and (os6) is the usual rule for congruence. Rule (os7) models a step due to transitions of the system that do not involve the observer. The interactions of the system with non-deterministic observers are defined by rule (os8); notice that, by (os5), if $O = \checkmark$ and $O' = \mathsf{0}$, then O'' is discarded.

Example 1. Consider a scenario where process P acts as a proxy of a shared resource for a client (which are not explicitly represented):

$$R = \overline{\text{lock}} \downarrow \overline{\text{unlock}}.(\text{quit}.\overline{\text{unlock}}).$$

R interacts with the resource to acquire a lock. This action is associated to compensation $\overline{\text{unlock}}$ whose aim is to release the resource if an error interrupts the normal execution flow. The client is granted to use of the resource until she sends message quit. Finally the resource is released ($\overline{\text{unlock}}$). Consider the observer

$$O = \text{lock}.(\lightning\overline{\text{quit}}.\text{unlock}.\checkmark)$$

that checks if the resource, after having been acquired, is released in case of failure of the clients' request to quit (action $\lightning\overline{\text{quit}}$). Notably, for any Γ, the observed system $\Gamma \vdash R \parallel O$ does not pass the test since the compensation is discarded by rule (os1) and O never reaches state $\checkmark$. Observed system $\Gamma \vdash \langle R \rangle \parallel O$ instead is satisfactory since the compensation, installed by (os2), can release the resource. ◇

The set Comp of *computations* (ranged over by c) is the set of (possibly infinite) sequences of states $S_0 \parallel O_0, \cdots, S_n \parallel O_n, \cdots$ such that $S_i \parallel O_i \rightsquigarrow S_{i+1} \parallel O_{i+1}$ for each i.

3.2 Testing Equivalences for ATc

The basic elements of the testing theory are the notions of *successful* and *non-divergent* computation. Intuitively, a computation is successful if the test is passed (i.e., the corresponding observer halts with ✓). Non-divergent computations are successful computations that reach ✓ before the occurrence of an error. We now cast the basic notions of the testing theory to ATc observed systems.

Definition 1. *Let $O \searrow \checkmark$ stand for $O = \checkmark + O'$ for some observer O'.*

- $S \parallel O \in \mathsf{States}$ *is* successful *if* $O \searrow \checkmark$*;*
- $\Gamma \vdash P \parallel O \in \mathsf{States}$ *is* diverging *if* $P = \mathbf{C}[\mathtt{err}]$ *for a context* $\mathbf{C}[\boxtimes]$*;*
- $c \in \mathsf{Comp}$ *is* successful *if it contains a successful state,* unsuccessful *otherwise;*
- $c = S_0 \parallel O_0, S_1 \parallel O_1, \ldots, S_n \parallel O_n, \ldots$ diverges *if either c is unsuccessful or there is* $i \geq 0$ *such that* $S_i \parallel O_i$ *is diverging and* $O_j \not\searrow \checkmark$ *for* $j < i$.

As customary in testing theory, the possible outcomes of computations are defined in terms of *result sets*, namely (non-empty) subsets of $\{\top, \bot\}$ where $\bot$ and $\top$ denote divergence and non-divergence, respectively.

Definition 2. *The* result set of $S \parallel O \in \mathsf{States}$, $\mathfrak{R}(S \parallel O) \subseteq \{\top, \bot\}$*, is defined by*

- $\top \in \mathfrak{R}(S \parallel O) \iff$ *there is a successful* $c \in \mathsf{Comp}$ *that starts from* $S \parallel O$*,*
- $\bot \in \mathfrak{R}(S \parallel O) \iff$ *there is* $c \in \mathsf{Comp}$ *starting from* $S \parallel O$ *such that c is diverging.*

As in [4], we consider *may*- and *must*-preorders and the corresponding induced equivalences.

Definition 3. *Given a system S and an observer O, we say that*

$$S \textsc{ may } O \iff \top \in \mathfrak{R}(S \parallel O) \qquad \textit{and} \qquad S \textsc{ must } O \iff \{\top\} = \mathfrak{R}(S \parallel O)$$

We define the preorders $\sqsubseteq_{\mathbf{m}}$ *(may preorder) and* $\sqsubseteq_{\mathbf{M}}$ *(must preorder) on systems:*

- $S \sqsubseteq_{\mathbf{m}} S' \iff (S \textsc{ may } O \implies S' \textsc{ may } O)$*, for all observers*
- $S \sqsubseteq_{\mathbf{M}} S' \iff (S \textsc{ must } O \implies S' \textsc{ must } O)$*, for all observers.*

The two equivalences $\simeq_{\mathbf{m}}$ *and* $\simeq_{\mathbf{M}}$ *corresponding to* $\sqsubseteq_{\mathbf{m}}$ *and* $\sqsubseteq_{\mathbf{M}}$ *are defined as expected:* $\simeq_{\mathbf{m}} \; = \; \sqsubseteq_{\mathbf{m}} \cap \sqsubseteq_{\mathbf{m}}^{-1}$ *and* $\simeq_{\mathbf{M}} \; = \; \sqsubseteq_{\mathbf{M}} \cap \sqsubseteq_{\mathbf{M}}^{-1}$.

Recall that (*i*) may-testing enforces some *fairness* ensuring that divergence is not "catastrophic" provided that there is a chance of success and (*ii*) that must-testing corresponds to liveness as it requires all possible computations to be successful.

4 Testing Theory for ATc at Work

The following examples show how attributes influence the reconfiguration of transactional scopes and how this is captured by our testing framework.

Example 2. Consider the service s with body R defined in Example 1. Let Γ be an environment such that $R \in \Gamma(s, \mathtt{r})$ and $R \in \Gamma(s, \mathtt{rn})$, namely in Γ there are (at least) two providers for s with the same body R but supporting different attributes. Consider the two possible clients, both invoking s and then releasing the resource:

$$P_1 = \langle s \multimap \{\mathtt{r}\}.\overline{\text{quit}}\rangle \qquad \text{and} \qquad P_2 = \langle s \multimap \{\mathtt{rn}\}.\overline{\text{quit}}\rangle.$$

The different attributes associated to s generate two different behaviours from P_1 and P_2 upon invocation (i.e., activation of endpoint $R = \overline{\text{lock}} \downarrow \overline{\text{unlock}}.\text{quit}.\overline{\text{unlock}}$):

$$S_1 = \Gamma \vdash \langle \overline{\text{quit}} \mid \overline{\text{lock}} \downarrow \overline{\text{unlock}}.\text{quit}.\overline{\text{unlock}}\rangle \qquad \text{by rule } (\mathtt{s5})$$
$$S_2 = \Gamma \vdash \langle \overline{\text{quit}}\rangle \mid \langle \overline{\text{lock}} \downarrow \overline{\text{unlock}}.\text{quit}.\overline{\text{unlock}}\rangle \qquad \text{by rule } (\mathtt{s7}).$$

Remarkably, R runs in the same transactional scope of the invoker in S_1 (due to the attribute $\mathtt{r}$), while it runs in a different scope in S_2 (due to the attribute $\mathtt{rn}$). Now take observer $O = \text{lock}.\text{unlock}.\checkmark + \lightning\overline{\text{quit}}.\text{unlock}.\checkmark$ that checks that the resource is unlocked both in case of normal execution and failure.

Running S_1 in parallel with O, and S_2 in parallel with O would results, after the synchronisation on channel lock, respectively in the system

$$S_1' = \Gamma \vdash \langle \overline{\text{quit}} \mid \text{quit}.\overline{\text{unlock}}\rangle_{\overline{\text{unlock}}} \qquad \text{and} \qquad S_2' = \Gamma \vdash \langle \overline{\text{quit}}\rangle \mid \langle \text{quit}.\overline{\text{unlock}}\rangle_{\overline{\text{unlock}}}$$

running in parallel with the continuation $\text{unlock}.\checkmark + \lightning\overline{\text{quit}}.\text{unlock}.\checkmark$ of O. ◇

In Example 2 both S_1 MAY O and S_2 MAY O hold true. In fact, there is at least a successful computation in both scenarios, namely the one in which the client manages to send $\overline{\text{quit}}$ so that there is no failure. In this case of normal execution both systems pass the test. On the other hand, an observer can tell apart systems S_1 and S_2 if it causes the failure of $\overline{\text{quit}}$. In fact, S_1' the failure would trigger the compensation $\overline{\text{unlock}}$ whereas in system S_2' the observer would remain blocked after the failure since the compensation is installed in a different scope. It is immediate from the definitions in § 3.2 that S_1 MUST O holds and $\bot \in \mathfrak{R}(S_2 \parallel O)$, therefore S_2 MUST O does not hold.

Example 2 shows that, by specifying different transactional attributes we obtain different reconfiguration semantics (i.e., the scopes of the resulting systems are differently configured) which may lead to different behaviours when failures have to be handled and propagated. In general the behaviour of a system changes depending on how its processes are nested in transactional scopes, as shown in Example 3.

Example 3. Given any environment Γ, it is possible to find $P, R, Q \in \mathcal{P}$ such that (omitting Γ for simplicity):

$$\langle P \mid R\rangle_Q \not\sqsubseteq_{\mathbf{M}} \langle P\rangle_Q \mid \langle R\rangle \qquad \textit{e.g.}, \langle \overline{a} \mid \overline{b}\rangle_{\overline{c}} \not\sqsubseteq_{\mathbf{M}} \langle \overline{a}\rangle_{\overline{c}} \mid \langle \overline{b}\rangle \text{ with } O = \lightning b.c.\checkmark$$
$$\langle P\rangle_Q \mid \langle R\rangle \not\sqsubseteq_{\mathbf{M}} \langle P \mid R\rangle_Q \qquad \textit{e.g.}, \langle \overline{a}\rangle_{\overline{c}} \mid \langle \overline{b}\rangle \not\sqsubseteq_{\mathbf{M}} \langle \overline{a} \mid \overline{b}\rangle_{\overline{c}} \text{ with } O = \lightning a.b.\checkmark$$
$$\langle P\rangle_Q \mid \langle R\rangle_Q \not\sqsubseteq_{\mathbf{M}} \langle P \mid R\rangle_Q \qquad \textit{e.g.}, \langle \overline{a}\rangle_{\overline{c}} \mid \langle \overline{b}\rangle_{\overline{c}} \not\sqsubseteq_{\mathbf{M}} \langle \overline{a} \mid \overline{b}\rangle_{\overline{c}} \text{ with } O = \lightning a.b.\checkmark$$
$$\langle P \mid R\rangle_Q \not\sqsubseteq_{\mathbf{M}} \langle P\rangle_Q \mid R \qquad \textit{e.g.}, \langle \overline{a} \mid \overline{b}\rangle_{\overline{c}} \not\sqsubseteq_{\mathbf{M}} \langle \overline{a}\rangle_{\overline{c}} \mid \overline{b} \text{ with } O = \lightning b.c.\checkmark$$
$$\langle P\rangle_Q \mid R \not\sqsubseteq_{\mathbf{M}} \langle P \mid R\rangle_Q \qquad \textit{e.g.}, \langle \overline{a}\rangle_{\overline{c}} \mid \overline{b} \not\sqsubseteq_{\mathbf{M}} \langle \overline{a} \mid \overline{b}\rangle_{\overline{c}} \text{ with } O = \lightning a.b.\checkmark$$
$$\langle P \mid R\rangle_Q \not\sqsubseteq_{\mathbf{M}} \langle P\rangle_Q \mid \langle R\rangle_Q \qquad \textit{e.g.}, \langle \overline{a} \downarrow \overline{e}.\overline{d} \mid d.\overline{b}\rangle_{\overline{c}} \not\sqsubseteq_{\mathbf{M}} \langle \overline{a} \downarrow \overline{e}.\overline{d}\rangle_{\overline{c}} \mid \langle d.\overline{b}\rangle_{\overline{c}} \text{ with } O = \lightning b.e.\checkmark.$$

On the left-hand side of each case above we present a counter-example for that case, where the observer is satisfied for the first process and not for the second one. In words, transactional scopes do not commute with or distribute over parallel composition. ⋄

5 Concluding Remarks and Related Work

Building on ATc [1], we define a theory of testing to study reconfigurable SOC transactions in presence of failures. The proposed framework captures the interplay between the semantics of processes and their compensations, and the dynamic reconfiguration of transactional scopes due to the run-time invocation of new services.

Transactional attributes of EJB have been adapted to SOC transitions in [1] where ATc has been introduced. The primitives of ATc allow one to *determine* and *control* the dynamic reconfiguration of distributed transactions so to have consistent and predictable failure propagation. Also, in [1] it has been given a type system for ATc that guarantees absence of failures due to misuse of transactional attributes.

A comparison of the linguistic features of ATc wrt other calculi featuring distributed transactions has been given in [1]; StAC [3] and CJoin [2] possibly are the closest calculi to ATc as they feature arbitrarily nested transactions and separate process communication from error/compensation. CJoin offers a mechanism to merge different scopes but it is not offering the flexibility of the transactional attributes of ATc. To the best of our knowledge, none of the calculi proposed in literature has given a testing semantics (in [5] testing equivalence is given for the Join calculus but not adapted to Cjoin).

One of the limitations of our approach is the lack of link mobility à la π-calculus; the extension of our approach to a name passing calculus is left as future work. Other interesting extensions would be to allow the communication of attributes and a primitive enabling a service s to make a parametrized invocation of a service s' using the same attribute supported by s (recall that attributes are when services are published in containers). Also, the interplay of attributes with the behaviour of observed systems deserves further investigation as in some contexts it could be possible to inter-change the attributes obtaining the same observed behaviour.

References

1. Bocchi, L., Tuosto, E.: A Java inspired semantics for transactions in SOC. In: TGC 2010. LNCS. Springer, Heidelberg (2010) (to appear)
2. Bruni, R., Melgratti, H., Montanari, U.: Nested commits for mobile calculi: extending Join. In: Lévy, J.-J., Mayr, E., Mitchell, J. (eds.) IFIP TCS 2004, pp. 563–576. Kluwer, Dordrecht (2004)
3. Butler, M., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 87–104. Springer, Heidelberg (2004)
4. De Nicola, R., Hennessy, M.C.B.: Testing equivalences for processes. Theoretical Comput. Sci. 34(1–2), 83–133 (1984)
5. Laneve, C.: May and must testing in the Join-Calculus. Technical Report UBLCS-96-4, Department of Computer Science University of Bologna (1996)
6. Panda, D., Rahman, R., Lane, D.: EJB 3 in action. Manning (2007)
7. Sun Microsystems. Enterprise JavaBeans (EJB) technology (2009), http://java.sun.com/products/ejb/

Grouping Nodes in Wireless Sensor Networks Using Coalitional Game Theory

Fatemeh Kazemeyni[1,2], Einar Broch Johnsen[1],
Olaf Owe[1], and Ilangko Balasingham[2,3]

[1] Department of Informatics, University of Oslo, Norway
{fatemehk,einarj,olaf}@ifi.uio.no
[2] The Interventional Center, Oslo University Hospital, Institute of Hospital Medicine, University of Oslo, Norway
[3] Department of Electronics and Telecommunication, Norwegian University of Science and Technology, Trondheim, Norway
ilangkob@medisin.uio.no

Abstract. Wireless sensor networks are typically ad-hoc networks of resource-constrained nodes; in particular, the nodes are limited in power resources. It can be difficult and costly to replace sensor nodes, for instance when implanted in the human body. Data transmission is the major consumer of power, so it is important to have power-efficient protocols. In order to reduce the total power consumption in the network, we consider nodes which cooperate to transmit data. Nodes which cooperate, form a *group*. A mobile node may at some point be without a group, in which case it is desirable for the node to be able to join a group. In this paper we propose a modification of the AODV protocol to decide whether a node should join a given group, using coalitional game theory to determine what is beneficial in terms of power consumption. The protocol is formalized in rewriting logic, implemented in the Maude tool, and validated by means of Maude's model exploration facilities.

1 Introduction

A wireless sensor network (WSN) often contains hundreds or thousands of sensor nodes equipped with sensing, computing, and communication devices such as short-range communication devices over wireless channels. These nodes may be distributed over a large area; e.g., WSNs can do area monitoring for some phenomenon of interest. In such an application, the main goal of the WSN is to collect data from the environment and send it to a sink node. In many cases, WSNs are ad-hoc networks with mobile nodes which need to self-configure.

Sensor nodes are very small and often difficult to replace. This size limitation introduces challenges for the design and management of WSNs; in particular, restrictions in memory, power, and communication capacity need to be considered in order to improve the longevity of the nodes. Power restriction is the most remarkable of these constraints: The range of data transmission depends on the power used by the node. Reduced power consumption is an important

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 95–109, 2010.
© IFIP International Federation for Information Processing 2010

goal in the design of WSN protocols. Because data transmission is expensive, the management of communication between nodes plays a vital role for the power efficiency of these networks. *Cooperation* between sensor nodes can potentially reduce the total power consumed for data transmission in the whole network.

Grouping is a method to organize node cooperation in a WSN. A group of nodes has a leader which receives data from the group members and communicates with the outside of the group. Nodes which are close to each other, may in principle communicate using less power. By cooperating inside a group, the group's members can decrease their transmission power to a minimum and still reach the leader. However, if nodes do not have fixed locations, the network topology can change. Nodes should compute the most efficient way to communicate in the network. Consequently, the group structure of the network may need to evolve. In a self-organizing network, a new node may want to join a group and the group needs to decide whether to accept the node.

This paper proposes a protocol to decide whether a node should join a group. We assume that a group leader forms and manages its group in a way that is beneficial for the group's members, and that a node can transmit directly to the group leader using maximum transmission power. Our protocol uses coalitional game theory to decide on group extensions. Adapting the AODV routing protocol for ad-hoc networks [21], a utility function over the dynamically determined route from the new node to the leader decides whether it is beneficial in terms of power consumption that the new node joins the group. We develop a formal, executable model of the proposed protocol in rewriting logic [16]. The resulting model is validated using Maude [3], some initial results are presented here.

Related work. WSNs present interesting challenges for formal methods, due to their resource restrictions and radio communication. This has led to research on how to develop modeling languages or extensions which faithfully capture typical features of sensors; e.g., mobility, location, radio communication, message collisions. In addition, WSNs need communication protocols which take resource usage into account. There is a very active field of research on protocol design for WSNs. However, protocol validation is mostly done with simulation-based tools, using NS-2, OMNeT+, and extensions such as Castalia [22] and SensorSim [20].

Formal techniques are much less explored in the development and analysis of WSNs, but start to appear. Among automata-based techniques, the TinyOS operating system has been modeled as a hybrid automaton [5] and UPPAAL has been applied to the LMAC protocol [7] and to the temporal configuration parameters of radio communication [25]. The CaVi tool combines simulation in Castalia with probabilistic model checking [6]. A recent process algebra for active sensor processes includes primitives for, e.g., sensing [4]. A Creol extension for heterogeneous environments includes radio communication [12]. The Temporal Logic of Actions has been used for routing tree diffusion protocols [18].

Ölveczky and Thorvaldsen have shown how a rich specification language like Maude is well-suited to model WSNs, using Real-Time Maude to analyze the performance of the OGCD protocol [19]. Their approach has also been combined with probabilistic model-checking to analyze the LMST protocol [13]. We follow

this line of research and use Maude as a tool to develop a grouping protocol [14] for WSNs, applying coalitional game theory to estimate power consumption. Noncooperational game theory has been used to reduce the power consumption of sensor nodes, applying a utility function to find the Nash equilibrium [17, 24, 11]. Coalitional game theory is applied to reduce the power consumption in WSNs by [23], who propose a merge and split approach for coalition formation. They calculate the value of the utility function for every possible permutation of nodes and find groups with the best utility value. This is as far as we know the only previous work that uses coalitional game theory for grouping the sensor networks. In contrast, we develop and formalize a protocol which considers nodes which may need to join a new group without reorganizing the entire WSN.

Paper overview. Section 2 introduces WSNs and grouping and Section 3 presents coalitional game theory. Section 4 proposes a group membership protocol based on coalitional game theory. Section 5 briefly summarizes rewriting logic and Maude, used to develop a formal model of the protocol in Section 6 and for analysis of an example topology in Section 7. Section 8 concludes the paper.

2 Grouping the Sensor Nodes

A sensor network is typically a wireless ad-hoc network, in which the sensor nodes support a multi-hop routing algorithm. In these networks, communication between nodes is generally performed by direct connection (single-hop) or through multiple hop relays (multi-hop). Multi-hop ad-hoc wireless networks use more than one wireless hop to transmit information from a source to a destination.

When a large number of sensor nodes are placed in the environment, neighbor nodes may be very close to each other. In this case, the transmission power level for communication with a neighbor can be kept low. Since nodes can cooperate with each other to transmit data, multi-hop communication in sensor networks is expected to consume less power than the traditional single-hop communication [1]. Furthermore, multi-hop communication can effectively overcome some signal propagation effects experienced in long-distance wireless communication.

Wireless sensor nodes use routing protocols to communicate with each other and to find the path to the designated sink node (or nodes) in order to transmit the data that is sensed from the environment. In most of these protocols, the nodes broadcast their data to all nodes that are within their data transmission range. This range is determined by the power used for transmission. In general, sensor nodes use their maximum data transmission power to cover a larger area and reach more nodes, both for data transmission and for routing. For example, in the standard AODV protocol [21], a node that moves or enters the network broadcasts a routing request package (`hello`) with maximum power to find neighbors. Due to node mobility, a node may need a new routing path, so it rebroadcasts a routing request to its neighbors using maximum power.

Grouping is a method for cooperation between nodes, in which nodes belong to distinct groups [14]. When nodes form a group, they help each other to transfer

data in a more organized way. Each group has a group *leader*; i.e., a designated member which receives data from the group members and communicates with other group leaders in order to route the data to its destination. Inside the group, it is not always necessary for a node to use its maximum transmission power. Instead, the group members can decrease the power consumed for communication and use their minimum transmission power to reach the group leader. The leader should be chosen carefully and this role can be exchanged between group members at specific time intervals. In this paper we do not focus on the leader selection issue but rather on the group management.

Sensor nodes in the real world are not designed to directly support grouping. We therefore assume that nodes know nothing about the grouping process. In contrast, the group leaders are special nodes that process the information of the newly entered sensor nodes and decide about their possible group membership. The group formation could be done by using different techniques. In general, the grouping of nodes can be done based on special characteristics or distance. In the first case, a special correlation among the sensors should be found by using vector quantization [9]. For example, all the sensor nodes that have similar sensed data could be placed in one group. In the second case, the sensor nodes are formed in different groups based on the distance. With this technique, a node's location is the important factor for group formation, but to have a better grouping, other factors such as interference could also be considered for group formation. The location of the nodes can be determined using different methods, such as GPS.

The AODV Routing protocol. The Ad-hoc On-Demand Distance Vector (AODV) routing protocol [21] is a reactive protocol; i.e., routes are created at need. It uses traditional routing tables, one entry per destination, and sequence numbers to decide if the routing information is up-to-date and to avoid loops. Note that AODV maintains time-based states in each node; if a routing entry has not been used recently, it expires and the node's neighbors are notified. Route discovery is based on query and reply cycles, and route information is stored in all nodes along the route as routing table entries. The AODV protocol works as follows:

1. Nodes broadcast `hello` messages to detect and monitor links to neighbors.
2. The route discovery process starts when a node which requires a route to another node, broadcasts a `rreq` message.
3. If the neighbor which receives this message has no route entry for the destination, or this entry is not up-to-date, the `rreq` is re-broadcasted with an incremented hop count which shows the length of the path.
4. While the `rreq` message is broadcasting through the network, each node remembers the reverse path to the source node.
5. If the receiver node is the destination or it has a routing path to the destination with a sequence number larger or equal to that of `rreq`, a `rrep` message is sent back to the source. The route to the destination is established when a `rrep` message is received by the original source node.
6. A source node may receive multiple `rrep` messages with different routes. It then updates its routing entries if the information is new in the sense that the `rrep` has a greater sequence number.

3 Coalitional Game Theory

Game theory [8] can be used to analyze behavior in decentralized and self-organizing networks. Game theory models the actions and choice of strategies of self-interested players, in order to capture the interaction of players in an environment such as a communication network. A game consists of

- a set of *players* $\mathbf{N} = \{1, 2, ..., n\}$.
- an indexed set of *possible actions* $A = A_1 \times A_2 \times ... \times A_n$, where A_i is the set of actions of player i (for $0 < i \leq n$).
- a set of *utility functions*, one for each player. The utility function u assigns a numerical value to the elements of the action set A; for actions $x, y \in A$ if $u(x) \geq u(y)$ then x must be at least as preferred as y.

Game theory can be divided into noncooperative [2] and cooperative game theory [8]. Noncooperative game theory studies the interaction between competing players, where each player chooses its strategy independently and each player's goal is to improve its utility or reduce its cost [23]. In contrast, cooperative (or coalitional) game theory considers the benefit of all the players. In coalitional games, players choose the strategies to maximize the utility for all players. In these games, cooperating groups of players are formed, called coalitions. Coalitional games are useful to design fair, robust, and efficient cooperation strategies in communication networks [23].

In a coalitional game (N, v) with N players, the utility of a coalition is determined by a *characteristic function* $v : 2^N \rightarrow \mathbb{R}$ which applies to coalitions of players. For a coalition $S \subseteq N$, $v(S)$ depends on the members of S, but not on the structure of the players. Most coalitional games have *transferable utility* (TU); i.e., the utility of a coalition can be distributed between the coalition members according to some notion of fairness. However, for many scenarios a coalition's utility cannot be captured by a single real value, or rigid restrictions are needed on the distribution of the utility. These games are known as coalitional games with *nontransferable utility* (NTU). In an NTU game, the payoff for each player in a coalition S depends on the actions selected by the players in S. The core of the coalitional game (N, v) is the set of payoff allocations that guarantees that no player has an incentive to leave N to form another coalition [23].

4 A Protocol for Deciding Group Membership

Consider the grouping problem for wireless sensor networks as a coalitional game. The sensor nodes are the players and the game is concerned with whether a node should join a group or not. The goal is to reduce the total power consumption in the network, so we need a utility function which reflects the power consumed for data transmission and signal interference. The utility function proposed by Goodman *et al.* [10] appears to be a suitable choice when power consumption is an important factor of the model [15]:

$$u(P_j, \delta_j) = (\frac{R}{P_j})(1 - e^{-0.5\delta_j})^L. \quad (1)$$

When applying u to a node j, P_j is the power used for message transfer by j and δ_j is the signal to interference and noise ratio (SINR) for j. In addition, R is the rate of information transmission in L bit packets in the WSN. For simplicity, we assume that the SINR is fixed and the same for all the nodes.

Wireless sensor nodes can transfer data with different amounts of power. Let P^{max} denote the maximum transmission power and P_j^{min} the minimum power for each node j, such that $0 \leq P_j^{min} \leq P^{max}$. When a node j cooperates in a group, it uses P_j^{min} for message transmission, and otherwise P^{max}. Consider a network of nodes $N = \{1, \ldots, n\}$. When there is no coalition between nodes in N, we have

$$\sum_{j=1}^{n} u(P_j, \delta) = \sum_{j=1}^{n} u_j(P^{max}, \delta).$$

In contrast, if all the nodes in N cooperate, we have:

$$\sum_{j=1}^{n} u(P_j, \delta) = \sum_{j=1}^{n} u(P_j^{min}, \delta).$$

Observe that if this utility function were applied naively, it would always be beneficial for nodes to form a coalition, as the result of decision making is the same for every topology of the network and every group:

$$\sum_{j=1}^{n} u(P_j^{min}, \delta) > \sum_{j=1}^{n} u(P^{max}, \delta).$$

However, in reality all the cooperating nodes use power in order to transmit data to the group leader, so it is not sufficient to only consider the power consumption of the original sender of data in the utility function. Although each node uses its minimum power to transmit data, the node's total power usage depends on the number of messages it needs to transmit. Each node on the route between the original sender and the leader, needs to send its own data as well as the data that it has received from the previous node. In general, the power consumption for the intermediate nodes will increase. We modify the utility function (Formula 1) to capture the overall power usage needed to transmit the data from the node to the leader following a given path:

$$u(P_j, \delta_j) = \left(\frac{R}{\sum_{n \in RP_{j,Leader}} P_n^{min}}\right)(1 - e^{-0.5\delta_j})^L, \qquad (2)$$

where the set $RP_{j,Leader}$ contains all nodes in the routing path between node j and the leader. This utility function is similar to Formula 1 except that the power that is applied is the sum of the powers consumed by all the nodes in the routing path through which data is transmitted from the sender to the leader.

Using this utility function, the leader can decide about the membership of a new node more precisely and with more realistic estimations. The result of this utility function depends on the specific topology, so coalition is not always beneficial. Depending on the value calculated using Formula 2, node j will become

a member of the group or not. According to the utility function it is more beneficial for the node to follow a path through the group than to act individually, if the following formula holds:

$$u(P^{max}, \delta) < u(P_j, \delta_j) = (\frac{R}{\sum_{n \in RP_{j,Leader}} P_n^{min}})(1 - e^{-0.5\delta_j})^L.$$

We adapt the AODV protocol to find the cheapest routing path between the new node and the leader in terms of power usage. In AODV, each node uses its maximum power to send all its messages, including `hello`, `rreq`, and `rrep` messages. In order to faithfully capture the real environment, the leader and the node should be in the signal range of each other. Consequently, the node can communicate directly with the leader by using its maximum transmission power. However, inside the group it is sufficient to use the minimum power for data transmission. Consequently, we let nodes transmit messages with minimum power when running the AODV routing protocol. In this situation, AODV can only find the routing path between the new node and the group leader in the case where coalition is possible. However, the shortest path in terms of hops is not necessarily the cheapest path when the minimum transmission power of nodes may vary. Therefore, we modify the AODV protocol to accumulate the needed power consumption along the path instead of the number of hops. The details are given in Section 6.

5 Rewriting Logic and Maude

The formal model of the protocol is defined in rewriting logic (RL) [16] and can be analyzed using Maude [3]. A rewrite theory is a 4-tuple (Σ, E, L, R) where the signature Σ defines the function symbols, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. Rewrite rules apply to terms of given sorts. Sorts are specified in (membership) equational logic (Σ, E). When modeling computational systems, different system components are typically modeled by terms of suitable sorts defined in the equational logic. The global state configuration is defined as a multiset of these terms. RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules supplementing the equations which define the term language. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [16]. If rewrite rules apply to non-overlapping sub-configurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in RL. Conditional rewrite rules $t \longrightarrow t'$ **`if`** `cond` are allowed, where the condition `cond` is a conjunction of rewrites and equations that must hold for the main rule to apply. In Maude, equations are denoted by **`eq`**, labeled rules by **`rl`** `[name]`, conditional ones by **`ceq`** and **`crl`**, respectively, and variable names are capitalized. Given an initial state of a model,

Maude supports simulation as well as breadth-first search through reachable states and model checking of finite reachable states for desired properties.

6 Modeling and Validation

In this section, we define a formal model of the group membership protocol in rewriting logic. In the model, we assume that there is no message loss in the protocol and that the topology of the network consists of a fixed number of nodes, but nodes can move and messages do not expire.

A *system configuration* is a multiset of objects and messages inside curly brackets. Following RL conventions, whitespace denotes the associative and commutative constructor for configurations. The term ⟨`O` : `Node` | `Attributes`⟩ denotes a `Node` object, where `O` is the identifier and `Attributes` is a set of attributes of the form `Attr` : `X`. Here, `Attr` is an attribute name and `X` the associated value. `Node` objects have the following attributes: the Boolean `leader?` indicates if the node is a leader, `leader` is a list that stores the information of the node's leader, the set `neighbors` contains the neighbor nodes, and the set `members` contains the group members of a leader node. The natural numbers `xLoc` and `yLoc` contain the horizontal and vertical position of the node, respectively. The natural number `power` stores the current sending power level of the node. The `routes` is a list of routes; i.e., of lists consisting of a destination ID, the next node in the path to the destination and the accumulated power that is required to send data to the destination. The natural number `reqid` stores the identifier of the last received message and `pred` the identity of the last neighbor that communicated with the node (the predecessor in the protocol).

We now explain the rules and equations modeling wireless message passing, node movement, the routing protocol, and the evaluation of the utility function.

6.1 Unicast and Broadcast

Unicast messages have the form `(M` **`from`** `O X Y P` **`to`** `O')` where `M` is the messages body (possibly with parameters), `O` the source with current location `(X,Y)`, `O'` the destination, and `P` the sending power used. A message will not reach its destination unless it is within the range. This is modeled by the equation

```
ceq (M from O X Y P to O´) ⟨O´: Node|xLoc: X´, xLoc: Y´, A⟩
  = ⟨O´: Node|xLoc: X´, xLoc: Y´, A⟩ if not inrange(X,Y,X´,Y´,P).
```

where `inrange` is a Boolean function checking that the two locations `(X,Y)` and `(X',Y')` are in range of each other with power `P` (using the calculated distance and the network parameters including the interference level). Note that this equation removes a message which cannot reach its destination, depending on the location values at sending time.

Transmission is modeled by a rule which creates a message, the equation above (if enabled) and a rule consuming the message if it is in range. This reflects the time when a message is queued for sending, the transmission time (immediate for wireless communication) and the queued at the destination, and the time the

message is taken out of the destination queue. Thus it models the immediate transmission mode of wireless communication, and it allows several nodes to move around at the same time.

Multicasting is modeled by allowing a multiset of destinations and the following equations which expand the destination list:

```
eq (M from O X Y P to noneOids) = none .
eq (M from O X Y P to O'; Os) =
    (M from O X Y P to O') (M from O X Y P to Os).
```

Here, `Os` denotes a multiset of object identities (with ";" as multiset constructor).

Wireless broadcasting uses messages `(M from O X Y P to all)` where `all` is a constructor indicating that it is sent to all nodes within range. The use of `all` is defined by the equation

```
eq {C (M from O X Y P to all)} = {C (M from O X Y P to nodes(C)-O)} .
```

Here, `nodes(C)` gives the multiset of all node identities in the configuration `C`. Thus broadcasting is reduced to multicasting. Due to the first equation above only nodes in the range can receive the message. This equation models the underlying network and therefore applies to the whole system.

6.2 Node Movements

In most WSNs, nodes can move and change their location. Therefore, a WSN model should provide suitable rules for changing the position of nodes. We have modeled three different methods for node movement. In the first method, the node can move freely everywhere, captured by rules that non-deterministically change the location of the node. In the second one, a node can move directly to a desired location. A rule will change the location of the node in one step. In the last method, a node can move to a desired location through a non-deterministic path, there are rules that determine the final destination and do non-deterministic but finite steps toward it. In each step, the vertical or horizontal location of the node is decreased or increased by one unit, but rules always check that the new location is closer to the desired location before the node moves.

6.3 The Regrouping

Each node should inform neighboring leader nodes about its movements. This is done by broadcasting a `hello` message with maximum power when it has changed position. The following rule represents the `hello` broadcasting:

```
rl [moving-done] :(movemsg Xn Yn from O X Y P to O)
⟨O :Node | xLoc: Xn, yLoc: Yn, neighbors: N, power: P, A⟩
⟶ ⟨O :Node | xLoc: Xn, yLoc: Yn, neighbors: N, power: P, A⟩
(hello from O X Y Pmax to all) .
```

When a neighboring group leader receives this `hello` message, a new node has entered the group's signal range. The leader will then start the process to decide whether the new node should be accepted as a group member based on the power information in the result path. The leader first runs the modified AODV protocol

(presented below in Section 6.4) with minimum power to find the cheapest path to the new node. If a path is found, the modified AODV protocol ends by letting the leader send a message `membershipMsg` to itself. This message starts the decision making process about the node's membership, which is captured by the following rules:

```
crl [Membershipdecision] :(membershipMsg Oc from O' X' Y' P' to O)
⟨O :Node | leader?: true, members: Os, xLoc: X, yLoc: Y, routes: RT,
      power: P, A⟩ ⟶
⟨O :Node | leader?: true, members: (Oc ; Os), xLoc: X, yLoc: Y,
      routes: RT, power: P, A⟩ (join from O X Y P to Oc)
if joinGroup(findPower(RT)).

rl [Join] :(join from O X Y P to Oc) ⟨Oc: Node | leader: [ O' X' Y' ], A⟩
⟶ ⟨Oc: Node | leader: [ O X Y ], A⟩ .
```

where the function `findPower` extracts the value of required power for data transmission from the routing table, and the function `joinGroup` represents the computation of the utility function (Formula 2), formalized as follows:

```
op joinGroup : Nat → Bool .
eq joinGroup (P)  = (RATE quo P * ((1 - 2.71 ^ (0.5 * I)) ^ PACK)
              > (RATE quo Pmax) * ((1 - 2.71 ^ (0.5 * I)) ^ PACK).
```

Here, `P` is the total power consumed in the routing path, and `Pmax`, `I`, `RATE`, and `PACK`, are constants reflecting the maximum sending power, the signal interference level, the transmission rate, and the packet size, respectively. These constants can be seen as network parameters, and suitable values given as parameters to the initial configuration. The output of `joinGroup` is a Boolean value. The leader uses this function to decide if a new node should be added as a member. The `Join` rule may be adjusted to enable optimal selection in case of multiple group membership offers.

6.4 The Routing Protocol

The routing protocol discussed in Section 4 is now formalized. The main difference between our protocol and AODV is that we find the *cheapest path* instead of the shortest one. In the model, each node has its own routing table that stores the path to each destination. For each destination, the routing table stores the following information: the next node on the path to the destination and the required power to send data to the destination. When the node finds a cheaper path to a destination (a path which requires less power), it updates its routing table and replaces the old path with the cheaper one. The neighbors of a node are stored in a list `neighbors`. Fig. 2 shows a simplified graph of our model.

All the messages in the routing protocol are modeled as messages in Maude and behave as explained in Section 6.1. The rules in Fig. 1 control the message propagation in the model by receiving a route request or a route reply message and sending a new message which is either a reply or a request. The rule `dest-rec-rreq` is enabled when the node that has received the request is the final destination. The next two rules, both named `rec-rreq`, are related and require that the receiver of the request message is not the destination. In

```
crl [dest-rec-rreq] :(singlerreq SID DID ID P from O X Y Pj to O')
⟨O' :Node|reqid: I, power: POW, xLoc: X', yLoc: Y', A⟩
⟶⟨O' :Node|reqid: ID, power: POW, xLoc: X', yLoc: Y', A⟩
(rrep SID DID ID POW from O' X' Y' POW to O) if O' = DID .

crl [rec-rreq1] :(singlerreq SID DID ID P from O X Y Pj to O')
⟨O' :Node|pred: O1, neighbors: N, reqid: I, power: POW, xLoc: X',
      yLoc: Y', A⟩
⟶⟨O' :Node|pred: O, neighbors: N, reqid: ID, power: POW, xLoc: X',
      yLoc: Y', A⟩ (rreq SID DID ID (P +POW) from O' X' Y' POW to N)
if (O' ≠ DID) ∧ (I < ID) .

crl [rec-rreq2] :(singlerreq SID DID ID P from O X Y Pj to O')
⟨O' :Node|pred: O1, neighbors: N, reqid: I, A⟩
⟶⟨O' :Node|pred: O, neighbors: N, reqid: ID, A⟩
if (O' ≠ DID) ∧ (I ≥ID) .

crl [rec-rrep1] :(rrep SID DID ID P from O X Y Pj to O')
⟨O' :Node|routes: RT, pred: O1, neighbors: N, power: POW,
      xLoc: X', yLoc: Y', A⟩
⟶⟨O' :Node|routes: RT, pred: O1, neighbors: N, power: POW, xLoc: X',
      yLoc: Y', A⟩ (rrep SID DID ID (P +POW) from O' X' Y' POW to O1)
if (O' ≠ SID) ∧ (findPower(RT, DID) > P) .

crl [rec-rrep2] :(rrep SID DID ID P from O X Y Pj to O')
⟨O' :Node|routes: (RT [ DID X Y ] RT'), pred: O1, neighbors: N,
      power: POW, xLoc: X', yLoc: Y', A⟩
⟶⟨O' :Node|routes: (RT [ DID O P ] RT'), pred: O1,
      neighbors: N, power: POW, xLoc: X', yLoc: Y', A⟩
(rrep SID DID ID (P +POW) from O' X' Y' POW to O1) if (O' ≠ SID)
∧ (findPower((RT [ DID X Y ] RT'), DID) < P)∧(findPower(RT, DID) ≠ 0).

crl [rec-rrep3] :(rrep SID DID ID P from O X Y Pj to O')
⟨O' :Node|routes: RT, pred: O1, neighbors: N, power: POW,
      xLoc: X', yLoc: Y', A⟩
⟶ (rrep SID DID ID (P +POW) from O' X' Y' POW to O1)
⟨O' :Node|routes: (RT [ DID O P ]), pred: O1,
      neighbors: N, power: POW, xLoc: X', yLoc: Y', A⟩
if (O' ≠ SID) ∧ (findPower(RT, DID) < P) ∧ (findPower(RT, DID) = 0) .

crl [src-rec-rrep] :(rrep SID DID ID P from O X Y Pj to O')
⟨O' :Node|routes: RT, pred: O1, neighbors: N, power: POW,
      xLoc: X', yLoc: Y', A⟩
 ⟶ (membershipMsg DID from O' X' Y' POW to O')
⟨O' :Node|routes: (RT [ DID O P ]), pred: O1, neighbors: N,
      power: POW, xLoc: X', yLoc: Y', A⟩ if (O' = SID) .
```

Fig. 1. Rules for message propagation

the first rule, the request message's `ID` is greater than previously seen by the receiver so the message is fresh. In the second rule the message has been seen before because the `ID` of the request message is less than the current message `ID`, so the message will be ignored. The three following rules represent the situation that the receiver of the reply message is not the original sender of the request. Note that in these rules the estimated power consumption is accumulated in the request messages by `P + POW`. In rule `rec-rrep1`, the routing table

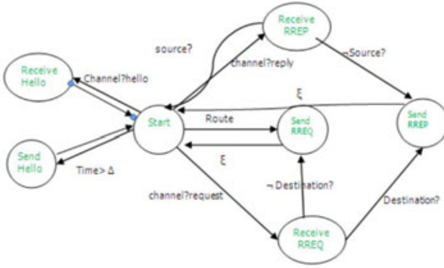


Fig. 2. The graph of the model

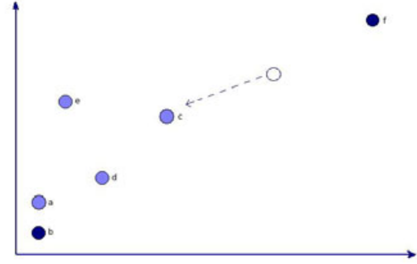


Fig. 3. The topology of the case study

remains unchanged because the current route to the destination is cheaper (the smaller power value in the table). Rule `rec-rrep2` changes the existing row in the routing table because the new path to the destination is cheaper than the the current one. In rule `rec-rrep3`, no previous path to the destination exists. Therefore, a new row is added to the routing table. The rule `src-rec-rrep` is enabled when the original sender of the request message has received the reply message. This rule sends a message `membershipMsg` that enables a rule that make decisions about node's membership in the group (see Section 6.3).

7 Analysis of the Case Study

Maude provides different tools for testing and validation of the model. It can run the model through just one path of the state space like a simulator (using the rewrite command `rew`), which gives us the result of a single run of the model. As a case study, we consider a topology with six nodes such that one of the nodes moves to the range of a group. The topology is shown in Fig. 3. In this topology, the nodes `b` and `f` are leaders of different groups. Nodes `a`, `d`, and `e` are the members of the group with leader `a`, and node `c` is the member of the group with leader `f`. We consider the scenario in which node `c` changes its location such that it comes within the range of the leader `b`. As group leader, node `b` decides about the membership of `c` in the group. The property that we expect from the system is beneficial membership; i.e., the membership of the node in the group is accepted by the leader `b` only when this membership is beneficial for the group. We first use Maude to check this property by simulating the model. The result of the simulation is given in Fig. 4. By inspecting the `members` attributes of leaders `b` and `f`, we see that node `c` now is a member of `b`'s group, while that of `f` is empty.

Simulation can not prove the correctness of the model because it just checks one path in the system's state space, to prove the validity of the model all the possible paths of the state space should be checked for failure. To prove the validity of the model, we search for all possible final states of our model using Maude's search command. The syntax `search initState =>! C:Configuration`

```
{⟨ "a" :Node | leader?: false, leader: [2 1 1], neighbors:
   ("b" ; "c" ; "d"), members: noneOids, xLoc: 1, yLoc: 1,
    routes: ([0 0 0] [3 3 1]), reqid: 1, pred: "d", power: 2 ⟩
⟨ "b" :Node | leader?: true, leader: [2 1 1], neighbors:
   ("a" ; "c" ; "d"), members: ("a" ; "c" ; "d" ; "e"), xLoc: 2,
    yLoc: 2, routes: ([0 0 0] [3 1 2]), reqid: 1, pred: "d", power: 2 ⟩
⟨ "c" :Node | leader?: false, leader: [2 2 2], neighbors:
   ("a" ; "e"), members: noneOids, xLoc: 6, yLoc: 8, routes:
    [0 0 0], reqid: 1, pred: "a", power: 2 ⟩
⟨ "d" :Node | leader?: false, leader: [2 1 1], neighbors:
   ("a" ; "b"), members: noneOids, xLoc: 4, yLoc: 3, routes:
   [0 0 0], reqid: 1, pred: "b", power: 1 ⟩
⟨ "e" :Node | leader?: false, leader: [2 1 1], neighbors:
   "c", members: noneOids, xLoc: 9, yLoc: 11, routes: [0 0 0],
    reqid: 0, pred: "a", power: 2 ⟩
⟨ "f" :Node | leader?: true, leader: [6 20 20], neighbors:
   "c", members: noneOids, xLoc: 9, yLoc: 11, routes: [0 0 0],
    reqid: 0, pred: "a", power: 2 ⟩ }
```

Fig. 4. Final state of the protocol for the case study

indicates that we search through all final states `C` reachable from the initial configuration `initState`. The result of this search is the same as for the simulation (cf. Fig. 4). Furthermore, the search shows that there is no other solution.

In order to validate the model, we use a correctness function. This is the function `joinGroup` that we used in the model to decide about membership of a node. We now search for a final state of the model such that node `c` is a member of the group of leader `b`, but such that the membership of this node is not beneficial for the group. In this case the function `joinGroup` should decide not to add the node to the group. We define a function `findCheapest` which statically finds the cheapest path from a node `j` to the leader in a given configuration `C`. Thus we compare the statically identified cheapest route with the result of running the protocol. If this search finds a state, it means that the model is not correct because a violation of the desired property has occurred. We do this analysis by a search command using a *such that* clause to specify the desired property.

```
search initState ⟶ ! {C:Configuration ⟨ "b" :Node | routes: RT:ListListNat,
members: ("c" ; ML:OidSet), ATTS:AttributeSet ⟩ } such that
  joinGroup (findCheapest (c,b, C:Configuration ⟨ "b" :Node | routes:
    RT:ListListNat, members: ("c" ; ML:OidSet), ATTS:AttributeSet ⟩ ))
      = false .
```

Maude checks all final states, but finds no solution to this search. This analysis demonstrates how Maude can be used to check the correctness of the protocol, limited to one initial state at the time. For a more in-depth analysis of the correctness of the protocol, we need to generate a set of initial states reflecting representative scenarios. However, how to generate such a set is beyond the scope of this paper and isleft for future work.

8 Conclusion

In this paper, we propose a protocol to decide on group membership for WSNs with mobile nodes. This work is done in the context of a cooperation project with the national hospital of Norway, where wireless technology is developed for medical applications. This paper addresses a subproblem which is common to several grouping protocols. Group members cooperate with each other to transmit data, in order to decrease the total power consumption of the group. When a node moves, it may need to join a new group and coalitional game theory is applied to find the best groups with respect to the total power consumption. The protocol we propose combines a modified version of the AODV protocol with coalitional game theory in order to find the cheapest route in a group with respect to power consumption. We have formalized the protocol in rewriting logic and used Maude to analyze its behavior for an example scenario.

In future work, we intend to refine the utility function used in this paper, in particular to capture the interference of the transmission signals. Other interesting extensions of our current work are criteria for the dynamic merging of groups and topologies in which the leader may change. For these problems, we intend to build on our current Maude model and to extend the model to capture real-time aspects of WSNs. Orthogonally to these extensions, we plan to strengthen our analysis by developing a broader base of representative WSN scenarios.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks 38(4), 393–422 (2002)
2. Başar, T., Olsder, G.J.: Dynamic non-cooperative game theory. SIAM, Philadelphia (1999)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science 285, 187–243 (2002)
4. Dong, J.S., Sun, J., Sun, J., Taguchi, K., Zhang, X.: Specifying and verifying sensor networks: An experiment of formal methods. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 318–337. Springer, Heidelberg (2008)
5. Ergen, S.C., Ergen, M., Koo, T.J.: Lifetime analysis of a sensor network with hybrid automata modelling. In: Raghavendra, C.S., Sivalingam, K.M. (eds.) Proc. First ACM Intl. Workshop on Wireless Sensor Networks and Applications (WSNA 2002), pp. 98–104. ACM, New York (2002)
6. Fehnker, A., Fruth, M., McIver, A.: Graphical modelling for simulation and formal analysis of wireless network protocols. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. LNCS, vol. 5454, pp. 1–24. Springer, Heidelberg (2009)
7. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and verification of the LMAC protocol for wireless sensor networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007)
8. Fudenberg, D., Tirole, J.: Game Theory. MIT Press, Cambridge (1991)
9. Gersho, A., Gray, R.M.: Vector Quantization and Signal Compression. Kluwer Academic Publishers, Norwell (1992)

10. Goodman, D., Mandayam, N.: Power control for wireless data. IEEE Personal Communications 7, 48–54 (2000)
11. Inaltekin, H., Wicker, S.B.: The analysis of nash equilibria of the one-shot random-access game for wireless networks and the behavior of selfish nodes. IEEE/ACM Trans. Netw. 16(5), 1094–1107 (2008)
12. Johnsen, E.B., Owe, O., Bjørk, J., Kyas, M.: An object-oriented component model for heterogeneous nets. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 257–279. Springer, Heidelberg (2008)
13. Katelman, M., Meseguer, J., Hou, J.C.: Redesign of the lmst wireless sensor protocol through formal modeling and statistical model checking. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 150–169. Springer, Heidelberg (2008)
14. Lloret, J., Palau, C.E., Boronat, F., Tomás, J.: Improving networks using group-based topologies. Computer Communications 31(14), 3438–3450 (2008)
15. Mackenzie, A.B., Wicker, S.B.: Game theory and the design of self-configuring, adaptive wireless networks. IEEE Communications Magazine 39(11), 126–131 (2001)
16. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
17. Miller, D.A., Tilak, S., Fountain, T.: Token equilibria in sensor networks with multiple sponsors. In: CollaborateCom. IEEE, Los Alamitos (2005)
18. Nair, S., Cardell-Oliver, R.: Formal specification and analysis of performance variation in sensor network diffusion protocols. In: Balsamo, S., Chiasserini, C.-F., Donatiello, L. (eds.) Proc. 7th Intl. Symp. on Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2004), pp. 170–173. ACM, New York (2004)
19. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theor. Comput. Sci. 410(2-3), 254–280 (2009)
20. Park, S., Savvides, A., Srivastava, M.B.: SensorSim: a simulation framework for sensor networks. In: Boukerche, A., Meo, M., Tropper, C. (eds.) Proc. 3rd Intl. Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2000), pp. 104–111. ACM, New York (2000)
21. Perkins, C.E., Belding-Royer, E.M.: Ad-hoc on-demand distance vector routing. In: WMCSA, pp. 90–100. IEEE Computer Society, Los Alamitos (1999)
22. Pham, H.N., Pediaditakis, D., Boulis, A.: From simulation to real deployments in WSN and back. In: Intl. Symp. on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2007), pp. 1–6. IEEE, Los Alamitos (2007)
23. Saad, W., Han, Z., Debbah, M., Hjørungnes, A., Başar, T.: Coalitional game theory for communication networks: A tutorial. IEEE Signal Processing Magazine 26(5), 77–97 (2009); Special Issue on Game Theory
24. Strauss, R., Abedi, A.: Game theoretic power allocation in sparsely distributed clusters of wireless sensors (gpas). In: Guizani, M., Mueller, P., Fähnrich, K.-P., Vasilakos, A.V., Zhang, Y., Zhang, J. (eds.) IWCMC, pp. 1454–1458. ACM, New York (2009)
25. Tschirner, S., Xuedong, L., Yi, W.: Model-based validation of QoS properties of biomedical sensor networks. In: de Alfaro, L., Palsberg, J. (eds.) Proc. 8th Intl. Conf. on Embedded Software (EMSOFT 2008), pp. 69–78. ACM, New York (2008)

Forgetting the Time in Timed Process Algebra
Timeless Behaviour in a Timestamped World

Anton Wijs

Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands
A.J.Wijs@tue.nl

Abstract. In this paper, we propose the notion of partial time abstraction for timed process algebras, which introduces the possibility to abstract away parts of the timing of system behaviour. Adding this notion leads to so-called partially timed process algebras and partially timed labelled transition systems. We describe these notions, and generalise timed branching bisimilarity to partially timed branching bisimilarity, allowing the comparison of systems with partial timing. Finally, with several examples and a case study, we demonstrate how partial time abstraction can be a useful modelling technique for timed models, which can lead to rigorous minimisations of state spaces.

1 Introduction

For many systems, correct and relevant verification involves timing aspects. In order to specify such systems, timed versions of existing modelling paradigms such as process algebras, automata, and Petri nets have been developed. However, timing aspects can be very demanding; when comparing two timed systems, they are only deemed equal in behaviour if both can perform the same actions at the same time. Checking timed temporal properties can often only practically be done if we partition system behaviour into time regions [1,2]. In order to do so, [17] defined several *time-abstracting* bisimilarities, and [12] defined strong and weak *time-abstracted* equivalences. Little research has been done to consider *relaxation* of the timing aspects by *partially* removing timing from timed systems where it is not relevant for either the verification of a temporal property, or a comparison with another system.

Where time is concerned, modelling languages either incorporate it completely, or they do not incorporate it at all. If time is present in the language, it means that either all actions are supplied with a time stamp, or that there are additional timing actions, which implement some relative notion of time. In process algebras, the inclusion of timing has naturally led to timed labelled transition systems, and subsequently to timed bisimilarities. The overall experience in this field is that these notions are very complex; for instance, in [10], it turned out that the definition of a specific notion of timed branching bisimilarity was not an equivalence when considering an absolute continuous (or dense) time domain. This was fixed by changing the definition. One of the conclusions,

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 110–124, 2010.
© IFIP International Federation for Information Processing 2010

however, was that the new notion was very demanding of the state spaces to be compared. In practice, this may result in too few equalities of state spaces.

Another complication was raised in [16]. There, it is argued that the abstraction of parts of a system specification does not lead to a sufficient decrease of the size of the resulting state space. Action abstraction is a powerful tool in explicit model checking, which can be used to hide action labels which are not interesting considering the system property to check. The main advantage is that this hiding of actions, which results in so-called 'silent steps', often leads to a reduced state space, which can be analysed more practically. The solution offered in [16] is to use untimed, instead of timed, silent steps. Surely, this means that the results of action abstraction in untimed and timed process algebras are indistinguishable, and therefore timed state spaces can be reduced more effectively.

However, this introduces complications. E.g. in an absolute time setting, implicit deadlocks due to ill-timedness may be removed, thereby introducing erroneous traces. Reniers and Van Weerdenburg [16] recognise this, but accept it. Another observation is that in general, there is no reason to inseparably connect action abstraction with time abstraction.

In this paper, we work out the notion of partial time abstraction, independently from action abstraction for an absolute time setting. Our choice for this setting, however, does not imply that the proposed notions cannot be applied to relative timing. We present a partially timed branching (Ptb) bisimulation relation, which is general enough to minimise both internal behaviour and time-hidden behaviour.[1] It should be stressed that, whereas we stick to explicit state model checking in this paper, the described notions can be adapted to a symbolic model checking setting. In Section 2, we start by describing a basic timed process algebra, timed labelled transition systems, and timed branching bisimilarity, after which we move to a more general setting, giving rise to the notions of a partially timed labelled transition system and Ptb bisimilarity in Section 3. In this Section, we also define several time abstraction operators for a timed process algebra. We illustrate the use of the new notions with examples, and Section 4 presents a case study. Alternative approaches and other time settings are considered in Section 5. Finally, Section 6 discusses related work, and Section 7 contains the conclusions.

2 Preliminaries

Timed Labelled Transition Systems. Let $\mathcal{A}$ be a non-empty set of visible actions, and τ a special action to represent internal events, with $\tau \notin \mathcal{A}$. We use $\mathcal{A}_\tau$ to denote $\mathcal{A} \cup \{\tau\}$. The time domain $\mathbb{T}$ is a totally ordered set with a least element

[1] Throughout the text, we use both the terms *untimed* and *time-hidden*, the difference between them being that an untimed action is literally not timed, while a time-hidden action is an action where the timing cannot be observed, but it is still there. When reasoning about time-hidden behaviour, though, we often state that a time-hidden action can in principle be fired at any time, since we do not know its timing.

0. We say that $\mathbb{T}$ is *discrete* if for each pair $u, v \in \mathbb{T}$ there are only finitely many $w \in \mathbb{T}$ such that $u < w < v$.

We use the notion of timed labelled transition systems from [19], in which labelled transitions are provided with a time stamp. A transition (s, ℓ, u, s') expresses that state s evolves into state s' by the execution of action ℓ at (absolute) time u. Such a transition is presented as $s \xrightarrow{\ell}_u s'$. It is assumed that execution of transitions does not consume any time. To keep the definition of timed labelled transition systems clean, consecutive transitions are allowed to have decreasing time stamps (i.e. ill-timedness); in the semantics, decreasing time stamps simply give rise to an (immediate) deadlock (see Definitions 2 and 4). To express time deadlocks, the predicate $\mathcal{U}(s, u)$ denotes that state s can let time pass until time u. A special state $\surd$ represents successful termination, expressed by the predicate $\surd\downarrow$. For the remainder of this paper, variables u, v, etc. range over $\mathbb{T}$.

Definition 1 (Timed labelled transition system). *A* timed labelled transition system *(*TLTS*) [15] is a triple* $(\mathcal{S}, \mathcal{T}, \mathcal{U})$*, where:*

1. $\mathcal{S}$ *is a set of states, including a state* $\surd$ *to represent successful termination;*
2. $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathbb{T} \times \mathcal{S}$ *is a set of transitions;*
3. $\mathcal{U} \subseteq \mathcal{S} \times \mathbb{T}$ *is a* delay relation*, which satisfies:*
   - $\mathcal{U}(s, 0)$*;*
   - *For* $u > 0$*,* $\mathcal{U}(s, u)$ *iff there exist* $v \geq u \in \mathbb{T}, \ell \in \mathcal{A}, s' \in \mathcal{S}$ *such that* $\mathcal{T}(s, \ell, v, s')$*.*

Timed Branching Bisimulation. Van Glabbeek and Weijland [11] introduced the notion of a *branching bisimulation* relation for untimed LTSs. Intuitively, a τ-transition $s \xrightarrow{\tau} s'$ is invisible if it does not lose possible behaviour (i.e., if s and s' can be related by a branching bisimulation relation). Van der Zwaag [19] defined a timed version of branching bisimulation, adding time stamps of transitions and ultimate delays $\mathcal{U}(s, u)$. Fokkink, Pang, and Wijs [10] showed that it was not an equivalence in a dense time domain, and improved the definition.

For $u \in \mathbb{T}$, the reflexive transitive closure of $\xrightarrow{\tau}_u$ is $\Rightarrow_u$. With $s \Rightarrow_u s'$ we express that s' is reachable from s by a number of τ-transitions at time u.

Definition 2 (Timed branching bisimulation). *Assume a* TLTS $(\mathcal{S}, \mathcal{T}, \mathcal{U})$*. A collection* B *of binary relations* $B_u \subseteq \mathcal{S} \times \mathcal{S}$ *for* $u \in \mathbb{T}$ *is a* timed branching bisimulation *[10] if* $s\ B_u\ t$ *implies:*

1. *if* $s \xrightarrow{\ell}_u s'$*, then*
   i. *either* $\ell = \tau$ *and* $s'\ B_u\ t$*,*
   ii. *or* $t \Rightarrow_u \hat{t} \xrightarrow{\ell}_u t'$ *with* $s\ B_u\ \hat{t}$ *and* $s'\ B_u\ t'$*;*
2. *if* $t \xrightarrow{\ell}_u t'$*, then vice versa;*
3. *if* $s\downarrow$*, then* $t \Rightarrow_u t'\downarrow$ *with* $s\ B_u\ t'$*;*
4. *if* $t\downarrow$*, then vice versa;*
5. *if* $u \leq v$ *and* $\mathcal{U}(s, v)$*, then for some* $n \geq 0$ *there are* $t_0, \ldots, t_n \in \mathcal{S}$ *with* $t = t_0$ *and* $\mathcal{U}(t_n, v)$*, and* $u_0 < \cdots < u_n \in \mathbb{T}$ *with* $u = u_0$ *and* $v = u_n$*, such that for* $i < n$*,* $t_i \Rightarrow_{u_i} t_{i+1}$ *and* $s\ B_w\ t_{i+1}$ *for* $u_i \leq w \leq u_{i+1}$*;*
6. *if* $u \leq v$ *and* $\mathcal{U}(t, v)$*, then vice versa;*

Two states s and t are timed branching bisimilar at u, *i.e.* $s \leftrightarroweq_{tb}^{u} t$, *if there is a timed branching bisimulation B with $s\ B_u\ t$. States s and t are* timed branching bisimilar, *i.e.* $s \leftrightarroweq_{tb} t$, *if they are timed branching bisimilar at all* $u \in \mathbb{T}$.

Transitions can be executed at the same time consecutively. By case 1 (and 2) in Definition 2, the behaviour of a state at some point in time is treated like untimed behaviour. Case 3 (and 4) deals with successful termination. By case 5 (and 6), time passing in a state s is matched by a related state t with a "τ-path" where all intermediate states are related to s at all times during the delay.[2] For a more detailed treatment of timed branching bisimilarity, see [10].

A Basic Process Algebra. In the following, x, y are variables and s, t are process terms or $\surd$, with $\surd$ a special state representing successful termination.

We present a basic process algebra with alternative ('$+$'), sequential ('$\cdot$') and parallel ('$||$') composition, and action abstraction ('τ_I'), which we will use in this paper. It is based on the process algebras $\text{BPA}_{\rho\delta U}$ [3] and timed μCRL [15]. With $a^{@u}$, we express that action a can be fired at time u. We consider the following transition rules, where the synchronisation of two actions $a, b \in \mathcal{A}_\tau$ resulting in an action c is denoted by $a \mid b = c$. The deadlock process is δ, which cannot fire any action, nor terminate successfully. If two actions a and b should never synchronise, we define that $a \mid b = \delta$. Usually, there are no synchronisations of τ actions defined, hence $\forall a. \tau \mid a = \delta \wedge a \mid \tau = \delta$. The process term $\tau_I(x)$ behaves as x with all action labels appearing in $I \subseteq \mathcal{A}$ rewritten to τ. Finally, the process term $u \gg x$ limits x to those alternatives with a first action timed at least at u.

$$\frac{}{\surd\downarrow} \qquad \frac{}{a^{@u} \xrightarrow{a}_u \surd} \qquad \frac{x \xrightarrow{a}_u x'}{\begin{array}{c} x+y \xrightarrow{a}_u x' \\ y+x \xrightarrow{a}_u x' \end{array}} \qquad \frac{x \xrightarrow{a}_u \surd}{\begin{array}{c} x+y \xrightarrow{a}_u \surd \\ y+x \xrightarrow{a}_u \surd \end{array}} \qquad \frac{x \xrightarrow{a}_u x'}{x \cdot y \xrightarrow{a}_u x' \cdot y} \qquad \frac{x \xrightarrow{a}_u \surd}{x \cdot y \xrightarrow{a}_u u \gg y}$$

$$\frac{x \xrightarrow{a}_u \surd \quad v \leq u}{v \gg x \xrightarrow{a}_u \surd} \qquad \frac{x \xrightarrow{a}_u x' \quad v \leq u}{v \gg x \xrightarrow{a}_u x'} \qquad \frac{x \xrightarrow{a}_u x' \qquad \mathcal{U}(y,u)}{\begin{array}{c} x||y \xrightarrow{a}_u x'||y \\ y||x \xrightarrow{a}_u y||x' \end{array}} \qquad \frac{x \xrightarrow{a}_u \surd \qquad \mathcal{U}(y,u)}{\begin{array}{c} x||y \xrightarrow{a}_u y \\ y||x \xrightarrow{a}_u y \end{array}}$$

$$\frac{x \xrightarrow{a}_u x' \quad y \xrightarrow{b}_u y' \quad a|b=c\neq\delta}{x||y \xrightarrow{c}_u x'||y'} \qquad \frac{x \xrightarrow{a}_u \surd \quad y \xrightarrow{b}_u y' \quad a|b=c\neq\delta}{\begin{array}{c} x||y \xrightarrow{c}_u y' \\ y||x \xrightarrow{c}_u y' \end{array}} \qquad \frac{x \xrightarrow{a}_u \surd \quad y \xrightarrow{b}_u \surd \quad a|b=c\neq\delta}{\begin{array}{c} x||y \xrightarrow{c}_u \surd \\ y||x \xrightarrow{c}_u \surd \end{array}}$$

$$\frac{x \xrightarrow{a}_u x' \qquad a \in I}{\tau_I(x) \xrightarrow{\tau}_u \tau_I(x')} \qquad \frac{x \xrightarrow{a}_u x' \qquad a \notin I}{\tau_I(x) \xrightarrow{a}_u \tau_I(x')} \qquad \frac{x \xrightarrow{a}_u \surd \qquad a \notin I}{\tau_I(x) \xrightarrow{a}_u \surd} \qquad \frac{x \xrightarrow{a}_u \surd \qquad a \in I}{\tau_I(x) \xrightarrow{\tau}_u \surd}$$

$$\begin{array}{lllll} \mathcal{U}(\surd,0) & \mathcal{U}(a^{@u},v) \text{ if } v \leq u & \mathcal{U}(x+y,v) \Leftrightarrow \mathcal{U}(x,v) \vee \mathcal{U}(y,v) & \mathcal{U}(x \cdot y,v) \Leftrightarrow \mathcal{U}(x,v) & \mathcal{U}(v \gg x,v) \Leftrightarrow \mathcal{U}(x,v) \\ & \mathcal{U}(\delta^{@u},v) \text{ if } v \leq u & \mathcal{U}(x||y,v) \Leftrightarrow \mathcal{U}(x,v) \wedge \mathcal{U}(y,v) & \mathcal{U}(\tau_I(x),v) \Leftrightarrow \mathcal{U}(x,v) & \mathcal{U}(u \gg x,v) \text{ if } v \leq u \end{array}$$

We note that with this process algebra and absolute timing, recursion is possible if we parameterise recursion variables with the current time (see Section 4).

[2] Clearly, this last case is very demanding in a dense time domain, since the states need to be relatable at *all* times during the delay. One of the main reasons for introducing partial timing, as introduced later in this paper, is therefore to alleviate this requirement in specific situations (see Example 1).

3 Towards Partial Timing

The main idea of partial timing is the ability to ignore exact timing of parts of a system if only action orderings are important in those parts for a verification task or comparison of systems. It is, therefore, similar to action abstraction, by which it is possible to hide action labels not important for a verification task. We prefer the possibility to hide timing aspects independently from hiding action labels, unlike [16], because first of all, timed τ-steps may be useful in practice, and second of all, likewise, time-hidden labelled actions can be very practical.

Next, we define a *partially timed labelled transition system* (PTLTS), i.e. an LTS which can incorporate timing information in specific parts. One issue is what the delayability of a state should be if it has time-hidden outgoing transitions. Time-hidden transitions cannot introduce time deadlocks, since time is irrelevant for them. However, from a process term, we do not wish to derive ill-timed traces (see Example 3). For this reason, we define a process term ultimate delay relation $\mathcal{U}$ and a state ultimate delay relation $\bar{\mathcal{U}}$. $\mathcal{U}$ is defined at the end of this Section, and $\bar{\mathcal{U}}$ is defined here. $\mathcal{U}$ in Def. 1 is extended to $\bar{\mathcal{U}}$ by expressing that time-hidden transitions have no influence on the delayability of the source state. In the following, $\mathbb{B}$ is the boolean domain, with elements T (true) and F (false).

Definition 3 (Partially timed labelled transition system). *A* partially timed labelled transition system *(*PTLTS*) is a triple* $(\mathcal{S}, \mathcal{T}, \bar{\mathcal{U}})$*, where:*

1. *$\mathcal{S}$ is a set of states, including a state $\surd$ to represent successful termination;*
2. *$\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathbb{T} \times \mathbb{B} \times \mathcal{S}$ is a set of transitions;*
3. *$\bar{\mathcal{U}} \subseteq \mathcal{S} \times \mathbb{T}$ is a* delay relation*, which satisfies:*
   - *$\bar{\mathcal{U}}(s, 0)$;*
   - *For $u > 0$, $\bar{\mathcal{U}}(s, u)$ iff there exist $v \geq u \in \mathbb{T}, \ell \in \mathcal{A}, s' \in \mathcal{S}$ such that $\mathcal{T}(s, \ell, v, \mathsf{T}, s')$.*

Transitions $(s, \ell, u, \mathsf{T}, s')$ express that state s evolves into state s' by execution of action ℓ at (absolute) time u. Transitions $(s, \ell, u, \mathsf{F}, s')$ express that state s evolves into state s' by the execution of action ℓ at some unobservable time.[3] It is assumed that the execution of transitions does not consume any time. A transition $(s, \ell, u, \mathsf{T}, s')$ is denoted by $s \xrightarrow{\ell}_u s'$, $(s, \ell, u, \mathsf{F}, s')$ is denoted by $s \xrightarrow{\ell}_{[u]} s'$. We write $s \xrightarrow{\ell} s'$ as a short-hand for $\exists v \in \mathbb{T}. s \xrightarrow{\ell}_{[v]} s'$ in case we are not interested in the time stamp. With $\xrightarrow{\ell}_{(u)}$, we denote the possibility to either perform $\xrightarrow{\ell}_u$ or $\xrightarrow{\ell}$, and $\Rightarrow_{(u)}$ denotes the reflexive transitive closure of $\xrightarrow{\tau}_{(u)}$, i.e. time-hidden and timed τ-steps can succeed each other. Finally, $s \Rightarrow^{(v)}_{(u)} s'$ denotes that there exist $s_0, \ldots, s_n \in \mathcal{S}$, for some $n \in \mathbb{N}$, with $s_n = s'$ and $u_0 <$

[3] Although the exact time of execution is given in $(s, \ell, u, \mathsf{F}, s')$, it will not be observable by the partially timed branching bisimilarity relation given later. We still keep the time stamp though, because we reason the timing is in fact still there and should have an influence on the (interaction of) system behaviour.

$\cdots < u_n \in \mathbb{T}$ with $u_0 = u$ and $u_n = v$, such that $s \Longrightarrow_{(u_0)} s_0 \Longrightarrow_{(u_1)} \cdots \Longrightarrow_{(u_n)} s_n$. If $\bar{\mathcal{U}}(s, u)$, then state s can let time pass until time u.

Now, we define a PTB bisimulation, which intuitively combines untimed and timed branching bisimilarity.

Definition 4 (Partially timed branching bisimulation). *Assume a* PTLTS $(\mathcal{S}, \mathcal{T}, \bar{\mathcal{U}})$. *A collection B of binary relations $B_u \subseteq \mathcal{S} \times \mathcal{S}$ for $u \in \mathbb{T}$ is a* partially timed branching bisimulation *if $s\ B_u\ t$ implies:*

1. *if $s \xrightarrow{\ell}_u s'$, then*
   - *i. either $\ell = \tau$ and $s'\ B_u\ t$,*
   - *ii. or $t \Longrightarrow_{(u)} \hat{t} \xrightarrow{\ell}_{(u)} t'$ with $s\ B_u\ \hat{t}$ and $s'\ B_u\ t'$;*
2. *if $t \xrightarrow{\ell}_u t'$, then vice versa;*
3. *if $s \xrightarrow{\ell} s'$, then*
   - *i. either $\ell = \tau$ and $s'\ B_u\ t$,*
   - *ii. or for some $v \geq u$, $t \Longrightarrow^{(v)}_{(u)} \hat{t} \xrightarrow{\ell} t'$ with $s\ B_v\ \hat{t}$ and $s'\ B_v\ t'$;*
4. *if $t \xrightarrow{\ell} t'$, then vice versa;*
5. *if $s \downarrow$, then $t \Longrightarrow_{(u)} t' \downarrow$ with $s\ B_u\ t'$;*
6. *if $t \downarrow$, then vice versa;*
7. *if $u \leq v$ and $\bar{\mathcal{U}}(s, v)$, then for some $n \geq 0$ there are $t_0, \ldots, t_n \in \mathcal{S}$ with $t = t_0$ and either $\bar{\mathcal{U}}(t_n, v)$ or $t_n \xrightarrow{\ell} t'$, and $u_0 < \cdots < u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $t_i \Longrightarrow_{(u_i)} t_{i+1}$ with $s\ B_w\ t_{i+1}$ for $u_i \leq w \leq u_{i+1}$;*
8. *if $u \leq v$ and $\bar{\mathcal{U}}(t, v)$, then vice versa.*

Two states s and t are PTB bisimilar at u, *denoted by $s \leftrightarroweq^u_{ptb} t$, if there is a* PTB *bisimulation B with $s\ B_u\ t$. States s and t are* PTB bisimilar, *denoted by $s \leftrightarroweq_{ptb} t$, if they are* PTB *bisimilar at all $u \in \mathbb{T}$.*

Cases 1 and 5 (and 2 and 6) directly relate to cases 1 and 3 (and 2 and 4) of Def. 2, respectively, the only difference being that time-hidden τ-steps are taken into account. Note that in 1.*ii*, a timed ℓ-step can be matched with a sequence of τ-steps followed by a *time-hidden* ℓ-step. We reason that a time-hidden ℓ-step can simulate a timed ℓ-step, because the first is not observably restricted by timing, while the latter is. The reverse is not true for the same reason. Besides, if we defined a timed ℓ-step and a time-hidden ℓ-step to be PTB bisimilar, then PTB bisimilarity would definitely not be an equivalence, which is undesired, since it would not be transitive. Consider $q \xrightarrow{\ell}_0 q'$, $r \xrightarrow{\ell}_1 r'$, and $s \xrightarrow{\ell} s'$ with $\ell \neq \tau$; clearly $q \not\leftrightarroweq_{ptb} r$, and therefore we either want $q \not\leftrightarroweq_{ptb} s$ or $r \not\leftrightarroweq_{ptb} s$. Case 3.*i* (4.*i*) is similar to 1.*i* (2.*i*), but dealing with a time-hidden τ-step. Case 3.*ii* (4.*ii*) states that a time-hidden ℓ-step can be matched by a sequence of τ-steps in which time may pass, leading to a state where a time-hidden ℓ-step is enabled. Case 7 (8) differs from case 5 (6) of Def. 2 in taking time-hidden τ-steps into account. Also, besides the possibility to match a delay by a τ-sequence to a state which can delay to the same moment in time, this latter state may just have the possibility to perform a time-hidden step (see Example 2). The idea behind

cases 3.*ii* (4.*ii*) and 7 (8) is that time-hidden steps are not observably subject to delays, i.e. the progress of time does not disable a time-hidden step. Consider the PTLTSs $s \xrightarrow{\tau}_2 s' \xrightarrow{a} s''$ and $t \xrightarrow{a} t'$. Here, $s \leftrightarroweq^0_{ptb} t$, since a delay of s to time 2 can be matched by t since it can perform a time-hidden step (case 7 (8) of Def. 4). Note that case 3 (4) is less demanding than 7 (8) concerning time; the first demands that states are related at the transition times, while the latter demands relations at all times. This is because in 3, a time-hidden step is fired, while in 7, s performs a delay, and is therefore time constrained.

Note that the union of PTB bisimulations is again a PTB bisimulation.

Theorem 1. *Timed branching bisimilar process terms P and Q are also* PTB *bisimilar, i.e.* $\leftrightarroweq_{tb} \subset \leftrightarroweq_{ptb}$.

Proof. A timed branching bisimulation relation B_u is a PTB bisimulation relation B'_u, since case 1.*i* (2.*i*) for B_u matches 1.*i* (2.*i*) for B'_u, 1.*ii* (2.*ii*) for B_u matches 1.*ii* (2.*ii*) for B'_u since $t \Rightarrow_{(u)} \hat{t} \xrightarrow{\ell}_{(u)} t'$ may consist of only timed steps. Cases 3 and 4 for B'_u are not applicable, since they concern time-hidden transitions. Case 3 (4) for B_u matches 5 (6) for B'_u since $t \Rightarrow_{(u)} t'$ may consist of only timed τ-steps. Finally, case 5 (6) for B_u matches 7 (8) for B'_u since $\bar{\mathcal{U}}(s,v) \Leftrightarrow \mathcal{U}(s,v)$ in the absence of time-hidden steps, and $\bar{\mathcal{U}}(t_n, v)$ is the only applicable condition for t_n. Furthermore, $t_i \Rightarrow_{(u_i)} t_{i+1}$ may only concern timed τ-steps. □

Similarly, we can prove that branching bisimilar terms are also PTB bisimilar. Furthermore, in [18], we prove that PTB bisimilarity is an equivalence relation.

In the following examples, $\mathbb{Q}_{\geq 0} \subseteq \mathbb{T}$.

Example 1. Consider the TLTSs $s_0 \xrightarrow{a}_{\frac{1}{3}} s_1 \xrightarrow{b}_1 s_2$ and $t_0 \xrightarrow{a}_{\frac{1}{3}} t_1 \xrightarrow{\tau}_{\frac{1}{2}} t_2 \xrightarrow{b}_1 t_3$. According to Def. 2, clearly $s_0\ B_u\ t_0$ for $u \geq 0$, and we need to determine that $s_1\ B_{\frac{1}{3}}\ t_1$. Since $\mathcal{U}(s_1, 1)$, by case 5 of Def. 2, we must check that we can reach a state t_i from t_1 via τ-steps, such that $\mathcal{U}(t_i, 1)$ and $s_1\ B_u\ t_i$ for some $u \geq \frac{1}{3}$. This means explicitly establishing that $s_1\ B_u\ t_1$ for all $\frac{1}{3} \leq u \leq \frac{1}{2}$ and $s_1\ B_u\ t_2$ for all $\frac{1}{2} \leq u \leq 1$. Moving to PTLTSs, $\mathcal{U}$ from Def. 1 and $\bar{\mathcal{U}}$ coincide, and case 7 of Def. 4 is similarly applicable. However, if we time-hide b, then by definition, $\bar{\mathcal{U}}(s_1, u)$ only for $u = 0$, hence case 7 is no longer a requirement. Note that if we only time-hide $t_2 \xrightarrow{b}_1 t_3$, the second PTLTS still simulates the first one.

Example 2. Consider the PTLTSs $s_0 \xrightarrow{a}_0 s_1 \xrightarrow{b}_{[3]} s_2$ and $t_0 \xrightarrow{a}_0 t_1 \xrightarrow{\tau}_1 t_2 \xrightarrow{b}_{[2]} t_3$. We have $s_0 \leftrightarroweq_{ptb} t_0$, since $s_0\ B_u\ t_0$ for $u \geq 0$, $s_1\ B_u\ t_1$ for $u \leq 1$, $s_1\ B_u\ t_2$ for $u \geq 0$, and $s_2\ B_u\ t_3$ for $u \geq 0$ is a PTB bisimulation.

In Example 2, the time-hidden b-step from s_1 can be matched with the delay $\bar{\mathcal{U}}(t_1, 1)$ (note in case 8 (7) of Def. 4 that the delay can simply be matched by s_1 by the fact that it can fire a time-hidden step), followed by the timed silent step from t_1 (by case 2.*i* (1.i)) and the time-hidden b-step from t_2 (by case 4 (3)). For time-hidden steps, there are no delay requirements, which greatly alleviates the demands of bisimilarity of timed systems when time-hiding is applied to them.

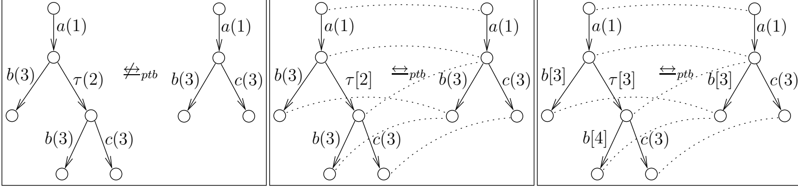


Fig. 1. Minimisation due to time abstraction

Like action abstraction, time abstraction may lead to the minimisation of LTSs, particularly when delayable actions are time-hidden. To illustrate this, in an untimed setting, $a{\cdot}(\tau{\cdot}(x + y) + x) = a{\cdot}(x + y)$ is an important axiom by which LTSs can be minimised using branching bisimilarity. Reniers and Van Weerdenburg [16] note that for a timed branching bisimilarity, a timed version of the axiom does not hold, and using untimed τ-steps reintroduces it. We show that in a partially timed setting, even a more general version of the axiom holds.

On the left of Figure 1, in which transitions $\xrightarrow{\ell}_u$ are written as $\xrightarrow{\ell(u)}$, the two PTLTSs are not PTB bisimilar, since concerning the states reached by taking the a-step, in the left PTLTS, at time 2 the decision can be made to take the b-step at time 3 by ignoring the τ-step, where in the right one, this cannot be done. In the middle of Figure 1, the τ-step is time-hidden, hence time does not influence the possibility to take the step, thus the two PTLTSs are PTB bisimilar, relatable states being connected with dashed lines. So far, this is similar to [16]. On the right of Figure 1, we have a PTLTS with a delayable action b; depending on whether the τ-step is performed or not, b may be executed at time 3 or 4. Such a situation is common in real system models. If we time-hide b, further reduction is possible. Note that it is not required to time-hide the silent step.

Note that time-hiding can alleviate the delay requirements of a (partially) timed branching bisimilarity, since time-hidden transitions are enabled immediately. E.g. at time u, we may fire a transition $s \xrightarrow{\ell}_v s'$ at $v \geq u$ after a delay to time v, while a transition $s \xrightarrow{\ell}_{[v]} s'$ can be fired immediately.

A Partially Timed Process Algebra. Next, we extend the process algebra of Section 2. With $a^{[@u]}$, we express that a can be fired at some hidden time. Keeping the exact timing when time-hiding is essential to ensure that additional system behaviour is not introduced. Consider the term $a^{@2}{\cdot}b^{@1}$. Surely b cannot be executed, since the time has already progressed too far. If we time-hide b, we do not want to lose this, since an observer might be unable to observe the timing of b, but in our opinion this does not mean that the timing is not there. The same argument holds for e.g. interleavings resulting from parallel compositions.

First, we define two versions of time abstraction, one for hiding timing information of specific action labels, the other for hiding a certain time interval.

Action-based Time Abstraction Transition Rules. In action-based time abstraction, timing information of all transitions with an action label in a given set $J \subseteq \mathcal{A}_\tau$ is hidden. The process term $\iota_J(P)$ hides timing of all actions of P in J.

$$\frac{x \xrightarrow{a}_u x' \quad a \in J}{\iota_J(x) \xrightarrow{a}_{[u]} \iota_J(x')} \qquad \frac{x \xrightarrow{a}_u \surd \quad a \in J}{\iota_J(x) \xrightarrow{a}_{[u]} \surd} \qquad \frac{x \xrightarrow{a}_u x' \quad a \notin J}{\iota_J(x) \xrightarrow{a}_u \iota_J(x')}$$

$$\frac{x \xrightarrow{a}_u \surd \quad a \notin J}{\iota_J(x) \xrightarrow{a}_u \surd} \qquad \frac{x \xrightarrow{a}_{[u]} x'}{\iota_J(x) \xrightarrow{a}_{[u]} \iota_J(x')} \qquad \frac{x \xrightarrow{a}_{[u]} \surd}{\iota_J(x) \xrightarrow{a}_{[u]} \surd}$$

Time-based Time Abstraction Transition Rules. All timing information between times u_1 and u_2 ($u_1, u_2 \in \mathbb{T}, u_2 \geq u_1$) are hidden. Such a time abstraction is useful if you want to hide the timing specifics of a certain time interval. The process term $\bar{\iota}_{u_1,u_2}(P)$ hides timing of all actions of P if $u_1 \leq u \leq u_2$.

$$\frac{x \xrightarrow{a}_u x' \quad u_1 \leq u \leq u_2}{\bar{\iota}_{u_1,u_2}(x) \xrightarrow{a}_{[u]} \bar{\iota}_{u_1,u_2}(x')} \qquad \frac{x \xrightarrow{a}_u \surd \quad u_1 \leq u \leq u_2}{\bar{\iota}_{u_1,u_2}(x) \xrightarrow{a}_{[u]} \surd} \qquad \frac{x \xrightarrow{a}_u x' \quad u_1 > u \vee u_2 < u}{\bar{\iota}_{u_1,u_2}(x) \xrightarrow{a}_u \bar{\iota}_{u_1,u_2}(x')}$$

$$\frac{x \xrightarrow{a}_u \surd \quad u_1 > u \vee u_2 < u}{\bar{\iota}_{u_1,u_2}(x) \xrightarrow{a}_u \surd} \qquad \frac{x \xrightarrow{a}_{[u]} x'}{\bar{\iota}_{u_1,u_2}(x) \xrightarrow{a}_{[u]} \bar{\iota}_{u_1,u_2}(x')} \qquad \frac{x \xrightarrow{a}_{[u]} \surd}{\bar{\iota}_{u_1,u_2}(x) \xrightarrow{a}_{[u]} \surd}$$

Additional Rules for Existing Operators. Looking back at the basic process algebra of Section 2, all rules except those for parallel composition can also straightforwardly be interpreted with $@u$ and $\xrightarrow{\ell}_u$ substituted by $[@u]$ and $\xrightarrow{\ell}_{[u]}$, respectively. Action $a^{[@u]}$ can then be seen as the time-hidden action a.

We define the following additional transition rules:

$$\frac{x \xrightarrow{a}_{[u]} x' \quad \mathcal{U}(y,u)}{\begin{array}{c} x||y \xrightarrow{a}_{[u]} x'||y \\ y||x \xrightarrow{a}_{[u]} y||x' \end{array}} \qquad \frac{x \xrightarrow{a}_{[u]} \surd \quad \mathcal{U}(y,u)}{\begin{array}{c} x||y \xrightarrow{a}_{[u]} y \\ y||x \xrightarrow{a}_{[u]} y \end{array}} \qquad \frac{x \xrightarrow{a}_{[u]} x' \quad y \xrightarrow{b}_u y' \quad a|b=c \neq \delta}{\begin{array}{c} x||y \xrightarrow{c}_u x'||y' \\ y||x \xrightarrow{c}_u y'||x' \end{array}}$$

$$\frac{x \xrightarrow{a}_{[u]} \surd \; y \xrightarrow{b}_u y' \; a|b=c \neq \delta}{\begin{array}{c} x||y \xrightarrow{c}_u y' \\ y||x \xrightarrow{c}_u y' \end{array}} \qquad \frac{x \xrightarrow{a}_{[u]} x' \; y \xrightarrow{b}_u \surd \; a|b=c \neq \delta}{\begin{array}{c} x||y \xrightarrow{c}_u x' \\ y||x \xrightarrow{c}_u x' \end{array}}$$

$$\frac{x \xrightarrow{a}_{[u]} \surd \; y \xrightarrow{b}_u \surd \; a|b=c \neq \delta}{\begin{array}{c} x||y \xrightarrow{c}_u \surd \\ y||x \xrightarrow{c}_u \surd \end{array}} \qquad \frac{x \xrightarrow{a}_{[u]} x' \quad y \xrightarrow{b}_{[u]} y' \quad a|b=c \neq \delta}{\begin{array}{c} x||y \xrightarrow{c}_{[u]} x'||y' \\ y||x \xrightarrow{c}_{[u]} y'||x' \end{array}}$$

$$\frac{x \xrightarrow{a}_{[u]} \surd \quad y \xrightarrow{b}_{[u]} y' \quad a|b=c \neq \delta}{\begin{array}{c} x||y \xrightarrow{c}_{[u]} y' \\ y||x \xrightarrow{c}_{[u]} y' \end{array}} \qquad \frac{x \xrightarrow{a}_{[u]} \surd \quad y \xrightarrow{b}_{[u]} \surd \quad a|b=c \neq \delta}{\begin{array}{c} x||y \xrightarrow{c}_{[u]} \surd \\ y||x \xrightarrow{c}_{[u]} \surd \end{array}}$$

To ensure that the parallel composition operator yields well-timed interleavings, the (process algebra related) ultimate delay relation $\mathcal{U}$ ignores time abstraction. Therefore, its definition from Section 2 is extended with $\mathcal{U}(a^{[@u]}, v)$ if $v \leq u$, $\mathcal{U}(\delta^{[@u]}, v)$ if $v \leq u$, $\mathcal{U}(\iota_J(x), v) \Leftrightarrow \mathcal{U}(x, v)$ and $\mathcal{U}(\bar{\iota}_{u_1,u_2}(x), v) \Leftrightarrow \mathcal{U}(x, v)$.

Example 3. Consider process terms $a^{[@1]} \mid\mid b^{@3}$ and $a^{[@2]} \cdot b^{@3}$. By definition, $\mathcal{U}(a^{[@1]}, u)$ for $u \leq 1$. This ensures that only $s_0 \xrightarrow{a}_{[1]} s_1 \xrightarrow{b}_3 s_2$ can be yielded from $a^{[@1]} \mid\mid b^{@3}$, i.e. the timing of a is not observable, but it is still there. On the other hand, $\bar{\mathcal{U}}$ ensures that $s_0 \xrightarrow{a}_{[1]} s_1 \xrightarrow{b}_3 s_2$ and $t_0 \xrightarrow{a}_{[2]} t_1 \xrightarrow{b}_3 t_2$ (yielded from $a^{[@2]} \cdot b^{@3}$) are PTB bisimilar, since both $\bar{\mathcal{U}}(s_0, u)$ and $\bar{\mathcal{U}}(t_0, u)$ only for $u = 0$.

We use $\mathcal{U}$ and $\bar{\mathcal{U}}$ for process terms and PTLTSs, respectively, since on the one hand, we do not want to enable extra orderings of actions when time-hiding, but on the other hand, we want to formalise that when comparing behaviour, delays do not apply for states from which only time-hidden transitions can be fired.

4 A Case Study: The Sliding Window Protocol

Now, we show the practical use of the techniques with a case study. For this, we extend the basic timed process algebra with guards, recursion, and data. Given a boolean condition b and process terms P, Q, a process term $P \triangleleft b \triangleright Q$ behaves as P iff $b = \mathsf{T}$, and as Q otherwise. Constructs can be parameterised with data, range over data domains, and incorporate recursion. For example, a process $P_t(d : \mathbb{D}) = \sum_{e:\mathbb{D}} a^{@t+t'}(e) \cdot P_{t+t'}(e)$, with d a variable of type $\mathbb{D}$, can execute actions a parameterised with any value of type $\mathbb{D}$, expressed using choice quantification [13,15], continuously, each time advancing the time by t' time units, and changing the state of P to the chosen value. Note that if $\mathbb{D}$ is an infinite data domain, this implies an impractical infinite number of alternatives. However, such a construct often represents receiving a message, and enforced synchronisation with a send action b then leads to a finite number of alternatives. Enforced synchronisation can be achieved with the *encapsulation* operator ∂_H, with $H \subseteq \mathcal{A}$. It disables firing the actions in the set H individually; only their synchronisation result can thus be fired. E.g. from $\partial_{\{a,b\}}(\sum_{n:\mathbb{N}} a(n) \mid\mid b(5))$, with $a \mid b = c$, we can only derive a transition labelled $c(5)$, since all options $a(n)$ with $n \neq 5$ cannot synchronise with $b(5)$ and are therefore blocked. Finally, $P_t = p \cdot P'_{t'}$ is a process which executes the process described by process term p, built using the operators of the timed process algebra, ending up in process $P'_{t'}$.

We do not yet have a real implementation of time abstraction and PTB bisimilarity, but we can obtain similar results for particular cases using the untimed μCRL [7] and LTSmin [8] toolsets. We stress, though, that this is far from ideal, and a real implementation is highly preferable. We model t as a data parameter of the process equations and actions (with $\mathbb{T} = \mathbb{Z}_{\geq 0}$), e.g. we model an action $a^{@t}(d)$, with d some data parameter, as $a(t, d)$. We have extended the μCRL LTS generator such that these modelled time stamps are correctly interpreted in parallel compositions (i.e. ensuring that e.g. $a(1, d) \mid\mid b(2, d')$ cannot yield a trace $s_0 \xrightarrow{b(2,d')} s_1 \xrightarrow{a(1,d)} s_2$), and developed a tool to time-hide a given set of actions in an LTS, i.e. to remove the time parameter from all occurrences of these actions, e.g. $\xrightarrow{a(t,d)}$ is rewritten to $\xrightarrow{a(d)}$ if a should be time-hidden. Furthermore, we use implementations of untimed strong and branching bisimilarity to minimise the LTSs. Please note that this approach has a number of limitations, among which:

1. We cannot use timed τ-steps, because the untimed bisimilarities cannot handle them. Time stamps on other steps are fine; since the time stamps are incorporated in the data parameters, they will be part of the comparison;
2. Time-hiding is applied on LTSs, hence globally. We cannot use time-hiding on e.g. specific occurrences of an action a. Furthermore, in practice, having to generate the full LTS before time-hiding can be applied can be a bottle-neck;
3. The progress of time should be ensured in the specification; time-deadlocks are not detected by the tools. This makes modelling quite hard;
4. We cannot use continuous time; in this Section, $\mathbb{T} = \mathbb{Z}_{\geq 0}$.

Next, we consider a timed *Sliding Window* (SW) protocol based on [9], which is a nice example of a complex, timed system. We use it to demonstrate how a PTLTS can be minimised for specific verification tasks. For a detailed explanation of the protocol, see [9]. A sender S needs to send a stream of data packets to a receiver R over an unreliable channel which may reorder, lose, and duplicate packets. The packets are labelled with sequence numbers modulo a given K, and they are processed in batches (windows) of size N. S may send new packets in its current window in the correct order, and resend any old packet in the window at any time. When R receives a packet for the first time, it is placed in the correct position in a buffer, and R sends an acknowledgement to S over an equally unreliable channel. R sorts the received packets on their sequence numbers, and, whenever possible, delivers some of the packets in the correct order on a channel as output. When S receives an acknowledgement, he moves his window forward beyond the acknowledged packet. An important correctness requirement is that $K \geq 2 * N$, and to ensure that sequence numbers are not reused too quickly, timing constraints are set; when S receives an acknowledgement for packet $K-1$, he waits $Lmax$ time units before sending a new packet 0 (constraint **CS**). R accepts packets with new numbers $Lmax$ time units after delivering a packet $K-1$ to the output (constraint **CR**) [9]. The channels nondeterministically delay the packets sent through them, and this aspect allows us to obtain smaller PTLTSs if we choose to abstract away the related timing. We illustrate this by providing part of the specification of S, and the entire specification of Fch, the channel between S and R, with $sa \mid ra = ca$:

$$
\begin{aligned}
&S(t:\mathbb{T}, \mathit{first}:\mathbb{N}, \mathit{ftsend}:\mathbb{N}, \mathit{tackmax}:\mathbb{N}, \mathit{nrm}:\mathbb{N}) = \\
&\quad sa(t+t', \mathit{ftsend})\cdot S(t+t', \mathit{first}, (\mathit{ftsend}+1)\ \mathbf{mod}\ K, \mathit{tackmax}, \mathit{nrm}-1) \\
&\qquad\qquad \lhd (\mathit{ftsend} = 0 \implies ((t+t') > (\mathit{tackmax} + \mathit{Lmax}) \wedge (\mathit{first} = \mathit{ftsend}))) \wedge \\
&\qquad\qquad \mathbf{inmod}(\mathit{ftsend}, \mathit{first}, \mathit{first}+N, K) \wedge \mathit{nrm} > 0 \rhd \delta + \\
&\quad \sum_{i:\mathbb{N}<K} sa(t+t', i)\cdot S(t+t', \mathit{first}, \mathit{ftsend}, \mathit{tackmax}, \mathit{nrm}) \\
&\qquad\qquad \lhd \mathbf{inmod}(i, \mathit{first}, \mathit{ftsend}, K) \wedge t < 30 \rhd \delta + \ldots
\end{aligned}
$$

$$
\begin{aligned}
&Fch(t:\mathbb{T}, L:(\mathbb{N},\mathbb{T})\mathbf{list}, \mathit{rbnd}:\mathbb{N}, \mathit{sbnd}:\mathbb{N}) = \\
&\quad \sum_{x:\mathbb{N}} \sum_{t_1:\mathbb{T}\geq t} ra(t_1, x)\cdot Fch(t_1, \mathbf{ins}((x, t_1 + (\mathit{Lmax}/2)), L), 3, \mathit{sbnd}) + \\
&\quad \sum_{x:\mathbb{N}} \sum_{t_1:\mathbb{T}\geq t} ra(t_1, x)\cdot Fch(t_1, \mathbf{ins}((x, t_1 + \mathit{Lmax}), L), 3, \mathit{sbnd}) +
\end{aligned}
$$

$$\sum_{x:\mathbb{N}} \sum_{t_1:\mathbb{T} \geq t} ra(t_1, x) \cdot Fch(t_1, L, rbnd - 1, sbnd) \triangleleft\ rbnd > 0 \triangleright \delta +$$
$$sb(\mathbf{headtime}(L), \mathbf{headnr}(L)) \cdot Fch(\mathbf{headtime}(L), \mathbf{tail}(L), rbnd, 3)$$
$$\triangleleft\ \mathbf{nonempty}(L) \wedge t \leq \mathbf{headtime}(L) \triangleright \delta +$$
$$sb(\mathbf{headtime}(L), \mathbf{headnr}(L)) \cdot Fch(\mathbf{headtime}(L), L, rbnd, sbnd - 1)$$
$$\triangleleft\ \mathbf{nonempty}(L) \wedge t \leq \mathbf{headtime}(L) \wedge sbnd > 0 \triangleright \delta$$

In S, t is the current time, *first* is the first packet in the current window, and *ftsend* is the next packet to send; *tackmax* equals the last time an acknowledgement for packet $K-1$ has been received, and *nrm* is the maximum number of new packets we want S to send. Special functions are written in boldface: **mod** is as usual, and **inmod** returns whether or not the first value lies between the second and the third value modulo the fourth value. The two options of S express sending new packets and resending old packets, respectively, where t' is a discrete jump forward in time (in our experiments, $t' = 1$) and i ranges over the old packets. Since $sa \mid ra = ca$, packets can be sent to the channel *Fch* if matching *sa* and *ra* are enabled. In *Fch*, besides the current time, we have a special list of tuples $(x : \mathbb{N}, t : \mathbb{T})$, with functions **ins**, which inserts a new tuple in the list ordered by increasing time t, and **headtime** and **headnr**, which return t and x of the head of the list, respectively; finally, **nonempty** determines whether the list is empty or not. Note that we use two bounds for receiving and sending packets, *rbnd* and *sbnd*, to ensure that packets are not continuously ignored by the channel; we reset these bounds to 3 after each successful receive and send.

The key observation can be done in the receive options of *Fch*: a packet can be stored in L with two time values: $t_1 + Lmax/2$ and $t_1 + Lmax$ (with t_1 the time of receiving the packet, and *Lmax* the maximum packet lifetime), and it can be ignored.[4] The time chosen here determines when the packet is sent with *sb* to R, which can fire action *rb* to receive packets. If we time-hide *sb* in the given specification part,[5] then traces can be 'folded' together if they concern the same packet numbers in the same order, i.e. we only care about the order in which the packets are sent, not at which moments in time. This can lead to rigorous PTLTS reductions; consider two traces $s_0 \xrightarrow{ca(0)}_0 s_1 \xrightarrow{ca(1)}_1 s_2 \xrightarrow{sb(0)}_{Lmax/2} s_3 \xrightarrow{sb(1)}_{(Lmax+1)/2} s_4$ and $t_0 \xrightarrow{ca(0)}_0 t_1 \xrightarrow{ca(1)}_1 t_2 \xrightarrow{sb(0)}_{Lmax} t_3 \xrightarrow{sb(1)}_{Lmax+1} t_4$. If we time-hide the *sb*-steps, these traces are PTB bisimilar (since $s_i \leftrightarrows^u_{ptb} t_i$ for $0 \leq i \leq 4$, $u \geq 0$), and hence can be reduced to one trace. The benefit of time-hiding here, is that even if we hide time entirely, time restrictions still apply; e.g. **CS** and **CR** still limit the potential behaviour of S. Note that as we allow more freedom in the packet delays, i.e. allow more than only two durations, the achieved reduction will increase.

Table 4 presents the size of an LTS generated using the μCRL toolset on 30 workstations in a distributed fashion, where each workstation had a quad-core INTEL XEON processor E5520 2.27 GHz, 24 GB RAM, and was running DEBIAN 2.6.26-19. We used the distributed reduction tool of LTSmin for the strong ($\leftrightarrows_s$) and branching ($\leftrightarrows_b$) bisimulation reductions. The different rows in

[4] In the real protocol, any time between t_1 and *Lmax* is possible.

[5] In the full specification, we actually time-hide the *cb*-transitions, where $sb \mid rb = cb$, and *sb* and *rb* are encapsulated.

Table 1. Sizes of LTSs describing the behaviour of a timed SW protocol

Description	**# States**	**# Transitions**
SW	335,236,894	1,940,674,714
$\leftrightarrows_s$ red. of $\iota_J(\mathrm{SW})$, $J = \mathcal{A} \setminus \{ca,\ cack2\}$	243,912,294	1,371,275,560
$\leftrightarrows_b$ red. of $\tau_J(\iota_J(\mathrm{SW}))$, $J = \mathcal{A} \setminus \{ca,\ cack2\}$	7,231,576	48,686,049
$\leftrightarrows_b$ red. of $\tau_I(\iota_{\mathcal{A}}(\mathrm{SW}))$, $I = \{ca\}$	234,398,155	1,310,272,020
$\leftrightarrows_b$ red. of $\iota_J(\tau_I(SW))$, $I = \mathcal{A} \setminus \{ca, deliver\}$, $J = \{\tau, ca\}$	7,450,689	49,944,368

Table 4 display the results of reductions taylored for checking specific properties: we get a full finite LTS using specification SW with values $N = 2, K = 5, nrm = 7$ and $Lmax = 2$, and bounding the overall time progress to 30 time units. For verification of constraint **CS** (row 2), we can time-hide all steps except the ones labelled *ca* and *cack2* (which represents S receiving an acknowledgement of a packet with number $K - 1$). Then, we can also still analyse the action ordering; if this is not needed, we can action-hide these actions, and get a better reduction (row 3). Similar reductions can be done for constraint **CR**. Row 4 represents a reduction useful for verifying action orderings other than *ca*; the latter label is completely hidden, all others are only time-hidden. Finally, hiding all except *ca* and *deliver* (the label denoting that R delivers a packet as output) allows us to check that the packets are delivered by R in the correct order, which is the most important property to check. We can also still analyse the timing of packet delivery. The full specification plus experiment instructions can be found at [18].

5 Some Timing Considerations

In this paper, we focus on absolute time. Here, we remark on applying partial timing with relative time. Consider the relative time process term $a^{@1} \cdot b^{@2}$. If we time-hide a, we lose the information when b actually starts executing, and the overall execution time. One course of action ([16]) is to maintain the overall duration, 'shifting' the timing of time-hidden actions towards the timed actions. If we follow this, then in our example, we get $a \cdot b^{@3}$. We advocate instead to keep the timing information as in our approach with absolute time, e.g. $a^{[@1]} \cdot b^{@2}$. Time shifting first of all does not guarantee that the overall duration is kept, e.g. if we time-hide b, this is lost anyway, and second of all, it does not seem entirely correct that time-hiding a has an influence on the duration of the execution of b. The way in which time progress is described in relative time should be reflected in PTB bisimilarity and the time-hiding operators. E.g. a sequence of τ-steps at time u in absolute time should be expressed as a sequence of τ-steps with time label 0 in relative time. Time-based time abstraction should be defined such that the current time is kept track of, to detect which actions should be time-hidden.

In the *two-phase* model, in which time progress is expressed using additional delay actions and transitions, partial timing seems to be achievable by action-hiding the delay actions; then, a branching bisimulation reduction results in their removal, in the absence of time non-determinism. This approach is straightforward when removing *all* timing; partial removal remains to be investigated.

Our next step is to investigate under which conditions a *rooted* [5,14] PTB bisimilarity is a congruence over our process algebra with time abstraction. In general this is not the case, because PTB bisimilar terms may 'interact' differently with other terms, since their timing may be partially ignored, but it is still there.

6 Related Work

In [4], a time free projection operator π_{tf} is used in a relative discrete time setting. A process $\pi_{tf}(P)$ is the process P made time free. In contrast to our work, they do not consider the possibility to only make certain action labels time free, or abstract away a specific period of time.

[16] considers untimed τ-steps, which allows better timed rooted branching bisimulation minimisation. There, ill-timedness of processes can 'disappear' when hiding actions. In our system, a time-hidden τ-step can be obtained from any action $a^{@u}$, by hiding both action label and timing, e.g. $\iota_{\{\tau\}}(\tau_{\{a\}}(a^{@u}))$. They do not consider the use of timed τ-steps and time-hidden labelled steps.

Approaches using *regions* and *zones* for timed automata [2] and *state classes* for Time Petri Nets [6] certainly have something in common with our work. There, the abstractions work system-wide, using a bisimilarity which fully ignores timing. We, however, abstract away some of the timing in the specification itself, and use a bisimilarity which does *not* ignore timing, but can handle time-hidden behaviour as well. [2] uses a function *Untime*, which removes the timing of system behaviour without introducing new action orderings. This is applied on the state space, though; no definition is given to abstract timing away from (parts of) the specification. In [17], several time-abstracting bisimilarities are defined for a relative time setting with a two-phase model. Like PTB bisimilarity, their strong time-abstracting bisimilarity preserves branching-time properties. Interestingly, it seems that, as suggested in Section 5, this coincides with untimed branching bisimilarity if we action-hide all the delay actions and no other actions, assuming there is no time non-determinism. They do not consider timed branching bisimilarity with time abstraction, nor hiding parts of the timing.

7 Conclusions

We proposed a generalisation of timed process algebras, TLTSs, and timed branching bisimilarity, by introducing time abstraction, by which we can hide the timing of system parts, and PTB bisimilarity, which is proven to be an equivalence relation. This allows to consider two timed systems equal in functionality, even if they are (partially) different in their timings. By hiding timing, no functionality is removed or added, i.e. no new orderings of actions are introduced. We have discussed how time abstraction, like action abstraction, can lead to minimised LTSs. The practical use of time-hiding has been demonstrated with a case study.

Future Work. We wish to fully implement the generalisations in a toolset. PTB bisimilarity can be analysed to see under which timing conditions it constitutes a

congruence, and we wish to investigate a complete axiomatisation of the process algebra and PTB bisimilarity, plus decidability of the latter.

Acknowledgements. We thank Wan Fokkink for his constructive comments.

References

1. Alur, R., Dill, D.: Automata for Modeling Real-Time Systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
2. Alur, R., Dill, D.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Baeten, J.C.M., Bergstra, J.A.: Real time process algebra. Formal Aspects of Computing 3(2), 142–188 (1991)
4. Baeten, J.C.M., Middelburg, C.A., Reniers, M.: A New Equivalence for Processes with Timing – With an Application to Protocol Verification. CSR 02-10, Eindhoven University of Technology (2002)
5. Bergstra, J.A., Klop, J.W.: Algebra of Communicating Processes with Abstraction. Theoretical Computer Science 37(1), 77–121 (1985)
6. Berthomieu, B., Diaz, M.: Modeling and Verification of Time Dependent Systems Using Time Petri Nets. IEEE Trans. on Softw. Engin. 17(3), 259–273 (1991)
7. Blom, S.C.C., Fokkink, W.J., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.C.: μCRL: A Toolset for Analysing Algebraic Specifications. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 250–254. Springer, Heidelberg (2001)
8. Blom, S.C.C., van de Pol, J.C., Weber, M.: Bridging the Gap between Enumerative and Symbolic Model Checkers. CTIT Technical Report TR-CTIT-09-30, University of Twente (2009)
9. Chkliaev, D., Hooman, J., de Vink, E.: Verification and Improvement of the Sliding Window Protocol. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 113–127. Springer, Heidelberg (2003)
10. Fokkink, W.J., Pang, J., Wijs, A.J.: Is Timed Branching Bisimilarity a Congruence Indeed? Fundamenta Informaticae 87(3/4), 287–311 (2008)
11. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3), 555–600 (1996)
12. Larsen, K., Wang, Y.: Time-abstracted bisimulation: Implicit specifications and decidability. Information and Computation 134(2), 75–101 (1997)
13. Luttik, S.P.: Choice Quantification in Process Algebra. PhD thesis, University of Amsterdam (2002)
14. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
15. Reniers, M.A., Groote, J.F., van der Zwaag, M.B., van Wamel, J.: Completeness of Timed μCRL. Fundamenta Informaticae 50(3-4), 361–402 (2002)
16. Reniers, M.A., van Weerdenburg, M.: Action Abstraction in Timed Process Algebra: The Case for an Untimed Silent Step. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 287–301. Springer, Heidelberg (2007)
17. Tripakis, S., Yovine, S.: Analysis of Timed Systems using Time-Abstracting Bisimulations. Formal Methods in System Design 18(1), 25–68 (2001)
18. Wijs, A.J.: Forgetting the Time in Timed Process Algebra - Appendix (2010), http://www.win.tue.nl/~awijs/timeabstraction/timeabs.html
19. van der Zwaag, M.B.: The cones and foci proof technique for timed transition systems. Information Processing Letters 80(1), 33–40 (2001)

Theory and Implementation of a Real-Time Extension to the π-Calculus*

Ernesto Posse and Juergen Dingel

School of Computing – Queen's University
Kingston, Ontario, Canada
{eposse,dingel}@cs.queensu.ca

Abstract. We present a real-time extension to the π-calculus and use it to study a notion of time-bounded equivalence. We introduce the notion of timed compositionality and the associated timed congruence which are useful to reason about the timed behaviour of processes under hard constraints. In addition to this meta-theory we develop an abstract machine for our calculus based on event-scheduling and establish its soundness w.r.t. the given operational semantics. We have built an implementation for a realistic language called kiltera based on this machine.

1 Introduction

The π-calculus [8] has become one of the most recognizable formal models of concurrency which allows the description of mobile processes. In order to model real-time mobile systems, a few process algebras have extended the π-calculus with an explicit notion of time including the $TD\pi$-calculus [11], the π_t-calculus [1], and the πRT-calculus [7]. In general, process algebras have been used to identify suitable notions of behavioural equivalence between processes to reason about their behaviour. In the context of real-time systems, time is essential to the comparison of system behaviours. Nevertheless, to the best of our knowledge, suprisingly little attention has been given to time-sensitive process equivalences for timed π-calculi.

Perhaps the most comprehensive study of timed equivalences for timed π-calculi are found in [3], [1], [6] and [2]. The first three study extensions to the π-calculus in which actions are associated with timers over discrete time. The fourth supports dense-time. In [3] and [6] some forms of timed barbed bisimilarity are studied, while [1] presents asynchronous bisimilarities and [2] explores some late bisimilarities. Nevertheless, these equivalences are quite stringent, as they require an exact match in the timing of the transitions of the processes being compared, *for all future behaviours*, and *as far in the future as the processes can run*. Real-time systems often are under *hard constraints* which require a system's response within a certain amount of time T. In this context all late responses are failures and therefore we can restrict our equivalence checking to *equivalence*

* This work has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and IBM Canada.

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 125–139, 2010.
© IFIP International Federation for Information Processing 2010

up-to time T. Furthermore, any reasonable equivalence must address the issue of compositionality: when is it safe to replace one process by another in a timed context? In this paper we define a time-bounded equivalence and show it to be compositional in our timed variant of the π-calculus.

The definition of the semantics of our calculus follows the standard approach and is given in terms of a Plotkin-style structural operational semantics (SOS). However, the purpose of our work is not only to study the theory of timed, mobile systems but also to provide a foundation for a realistic, executable, high-level modelling language for such systems. To this end, we define an abstract machine which complements the SOS of our calculus by describing execution at a level of abstraction more suitable for implementation. We prove the soundness of the machine w.r.t. the SOS. A distinguishing feature of our abstract machine is that it is based on *event-scheduling* as used in discrete-event simulation [15]. Event-scheduling does not iterate over all clock ticks whenever events are far apart in time, unlike the discrete-time approach used by existing implementations of timed π-calculi. We have validated the abstract machine via the implementation of the language kiltera [9] which has been used for teaching (in graduate courses at Queen's and McGill universities) and the modelling and analysis of complex systems such as automobile traffic simulation.

The contributions of this paper are: a process algebra that supports mobility and real-time with higher-level features such as pattern-matching; a formal operational semantics, including a new timed observational equivalence, the notion of timed compositionality and timed congruence; a sound abstract machine based on event-scheduling with a working implementation.

Paper organization: Section 2 introduces our calculus, its syntax, its operational semantics. Timed equivalence is studied in Section 3. Section 4 develops the abstract machine. Section 5 concludes. For proofs see [10] and [9].

2 Timed, Mobile Processes: The π_{klt}-Calculus

We define our timed π-calculus, which extends the asynchronous π-calculus with delays, time-value passing and unlike other variants, time observation and pattern-matching.

Definition 1. ***(Syntax)*** *The set $\mathcal{P}$ of π_{klt}* ***terms****, the set $\mathcal{E}$ of* ***expressions*** *and the set of* ***patterns*** *$\mathcal{F}$ are defined by the BNF below. Here P, P_i range over* process terms*, $x, y, ...$ range over the set of* ***(channel/event or variable) names****, A ranges over the set of* ***process names****, E ranges over* expressions*, and F ranges over* patterns*. Process definitions have the form: $A(x_1, ..., x_n) \stackrel{def}{=} P$. n ranges over floating point numbers, s ranges over strings, and f ranges over function names, with function definitions having the form: $f(x_1, ..., x_n) \stackrel{def}{=} E$, and the index set I is a subset $\{1, ..., n\} \subseteq \mathbb{N}$.*

$$\begin{array}{rl} P ::= & \surd \quad | \quad x!E \quad | \quad \sum_{i\in I} x_i?F_i@y_i.P_i \quad | \quad \nu x.P \\ & | \quad \Delta E.P \quad | \quad P_1 \parallel P_2 \quad | \quad A(x_1, ..., x_n) \end{array}$$

$$E ::= \varnothing \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \text{“}s\text{”} \mid x \mid \langle E_1, ..., E_m \rangle \mid f(E_1, ..., E_m)$$

$$F ::= \varnothing \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \text{“}s\text{”} \mid x \mid \langle F_1, ..., F_m \rangle$$

Expressions E are either constants ($\varnothing$ represents the *null* constant), variables (x), tuples of the form $\langle E_1, ..., E_m \rangle$ or function applications $f(E_1, ..., E_m)$. Patterns F have the same syntax as expressions, except that they do not include function applications.

The process $\surd$ simply terminates. The process $x!E$ is a *trigger*; it triggers an event x with the value of E. Alternatively, we can say that it sends the value of E over a channel x. The expression E is optional: $x!$ is shorthand for $x!\varnothing$. A process of the form $\sum_{i \in I} \beta_i.P_i$ is a *listener*, where each β_i is a *guard* of the form $x_i?F_i@y_i$. This process listens to all channels (or events) x_i, and when x_i is triggered with a value v that matches the pattern F_i, the corresponding process P_i is executed with y_i bound to the amount of time the listener waited, and the alternatives are discarded[1]. The suffixes F_i and $@y_i$ are optional: $x?.P$ is equivalent to $x?y@z.P$ for some fresh names y and z. The process $\nu x.P$ hides the name x from the environment, so that it is private to P. Alternatively, $\nu x.P$ can be seen as the creation of a new name, *i.e.*, a new event or channel, whose scope is P. We write $\nu x_1, x_2, ..., x_n.P$ for the process term $\nu x_1.\nu x_2....\nu x_n.P$. The process $\Delta E.P$ is a *delay*: it delays the execution of process P by an amount of time equal to the value of the expression E.[2] The process $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2. We write $\Pi_{i \in I} P_i$ for $P_1 \parallel \cdots \parallel P_n$. The process $A(y_1, ..., y_n)$ creates a new instance of a process defined by $A(x_1, ..., x_n) \stackrel{def}{=} P$, where the ports $x_1, ..., x_n$ are substituted in the body P by the channels (or values) $y_1, ..., y_n$.

Timeouts are obtained as a derived construct: the process term $(\sum_{i \in I} \beta_i.P_i) \stackrel{E}{\triangleright} Q$ represents a listener process with a timeout. If after an amount of time determined by the value of the expression E, none of the channels have been triggered, control passes to Q. We define this term as follows:

$$(\Sigma_{i \in I} \beta_i.P_i) \stackrel{E}{\triangleright} Q \stackrel{def}{=} \nu s.((\Sigma_{i \in I} \beta_i.P_i + s?.Q) \parallel \Delta E.s!)$$

The local event s can be thought of as the timeout event. Also, as in the asynchronous π-calculus [5], an output with a continuation $x!E.P$ is syntactic sugar

[1] Note that to enable an input guard it is not enough for the channel to be triggered: the message must match the guard's pattern as well. Pattern-matching of inputs means that the input value must have the same "shape" as the pattern, and if successful, the free names in the pattern are bound to the corresponding values of the input. For example, the value $\langle 3, \mathsf{true}, 7 \rangle$ matches the pattern $\langle 3, x, y \rangle$ with the resulting binding $\{^{\mathsf{true}}/_x, {}^7/_y\}$. The scope of these bindings is the corresponding P_i.

[2] The value of E is expected to be a non-negative real number. If the value of E is negative, $\Delta E.P$ cannot perform any action. Similarly, terms with undefined values (*e.g.*, $\Delta(1/0).P$) or with incorrectly typed expressions (*e.g.*, $\Delta\mathsf{true}.P$) cause the process to stop. Since the language is untyped we do not enforce these constraints statically.

for $x!E \parallel P$. Other useful extensions include the term $\mathsf{match}\, E\ \mathsf{with}\ F_1 \rightarrow P_1 \mid \cdots \mid F_n \rightarrow P_n$ which is syntactic sugar for $\nu x.(x!E \parallel x?F_1.P_1 + \cdots + x?F_n.P_n)$, and the conditional term $\mathsf{if}\, E\ \mathsf{then}\ P\ \mathsf{else}\ Q$ which is shorthand for $\mathsf{match}\ E\ \mathsf{with}\ \mathsf{true} \rightarrow P \mid \mathsf{false} \rightarrow Q$.

The suffix $@y_i$ of input guards is inherited from Timed CSP, but it is absent in all other timed variants of the π-calculus. This construct gives the calculus the power to measure the timing of events, and determine future behaviour accordingly.

An example: testing server response times. We illustrate the language with a short example. Consider a simple device to measure a server's response time to some query. To begin the test, the device (D) waits for a signal b from some client. The client provides a maximum response time t and four channels q, a, r, and m. The channels q and a are links to the server, where the device will send the query (q) and where it will expect the answer (a). The channel r is where the client expects to observe the response time. After sending a sample query to the server, the device waits for a response. If the server fails to respond within t seconds, the device will trigger a timeout event (m). We can model this testing system as follows:

$$D(b) \stackrel{def}{=} b?\langle t, q, a, r, m\rangle.q!.(a?@e.r!e.D(b)) \stackrel{t}{\triangleright} m!.D(b)$$

A model of a server, abstracting its internal execution, could be given by $S(q, a, u) \stackrel{def}{=} q?.\Delta u.a!.S(q, a, u)$. A client that uses D to test between two servers succesively and then decides to interact with the fastest is modelled as follows:

$$\begin{aligned} C(q_1, a_1, q_2, a_2, b) \stackrel{def}{=} \nu r_1, m_1, r_2, m_2.(&b!\langle 5, q_1, a_1, r_1, m_1\rangle \parallel b!\langle 5, q_2, a_2, r_2, m_2\rangle \\ &\parallel r_1?e_1.r_2?e_2.\,\mathsf{if}\ e_1 < e_2\ \mathsf{then}\ C'(q_1, a_1) \\ &\qquad \mathsf{else}\ C'(q_2, a_2) \\ &\parallel m_1?.C'(q_2, a_2) + m_2?.C'(q_1, a_1)) \end{aligned}$$

This client asks the testing device to test two servers (whose channels are parameters to the client). If both servers respond within 5 seconds (r_1 and r_2) the client selects the smaller response time and becomes C' which interacts with the corresponding server only. If it receives a timeout event for either server, it selects the other one. The complete system could be modelled as follows:

$$\nu q_1, a_1, q_2, a_2, b.\big(S(q_1, a_1, 3.2) \parallel S(q_2, a_2, 4.1) \parallel D(b) \parallel C(q_1, a_1, q_2, a_2, b)\big)$$

Operational semantics. We now define the semantics formally. Let $\mathcal{N}$ denote the set of all possible names (including channel names). Let $\mathcal{V}$ denote the universe of possible values including booleans, real numbers, strings, tuples of values and channel names, and $\mathcal{B} \subseteq \mathcal{V}$ is the set of basic constants (*i.e.*, non-tuple values). We write $\mathrm{n}(v)$ for the set of all channel names occurring in the value v. To simplify the presentation we assume we have a function $eval : \mathcal{E} \rightarrow \mathcal{V}$ that

given an expression returns its value.[3] A sequence of names or values $x_1, ..., x_n$ is abbreviated as $\tilde{x}$. We denote with $\text{fn}(P)$ the set of *free names* of P (*i.e.*, names not bound by either ν or an input guard). A *substitution* is a function $\sigma : \mathcal{N} \rightarrow \mathcal{V}$. We write $\{^{V_1}/_{x_1}, ..., ^{V_n}/_{x_n}\}$ or $\{^{\tilde{V}}/_{\tilde{x}}\}$ for the substitution σ where $\sigma(x_1) = V_1$, ..., $\sigma(x_n) = V_n$ and $\sigma(z) = z$ for all $z \notin \{x_1, \ldots, x_n\}$. We write $\text{dom}(\sigma)$ for $\{x_1, ..., x_n\}$. Furthermore, we write $\sigma[^V/_x]$ for substitution update[4]. Substitution is generalized to processes as a function $\sigma : \mathcal{P} \rightarrow \mathcal{P}$ in the natural way performing the necessary renamings to avoid capture of free names as usual. We write $P\sigma$ for $\sigma(P)$ denoting the process where all free occurrences of each x in σ have been substituted by $\sigma(x)$. We denote with $\mathcal{M}$ the set of all name substitutions. We write $P \equiv_\alpha Q$ if Q can be obtained from P by renaming of bound names. We use $\mathbb{R}_0^+$ to denote the non-negative reals.

Pattern matching is formally defined by a function $match : \mathcal{F} \times \mathcal{V} \times \mathcal{M} \rightarrow \mathcal{M} \uplus \{\bot\}$ which takes as input a pattern, a datum (*i.e.*, a concrete value) and a substitution and returns either a new substitution which extends the original substitution with the appropriate bindings, or $\bot$ if the datum does not match the pattern. The substitution provided as input is used to ensure that all occurrences of a variable in a tuple match the same data. For a formal definition of this function see [10].

Any well-defined semantics must ensure that processes which are structurally equivalent behave in the same way. We now define such an equivalence relation, called *structural congruence*.

Definition 2. ***(Structural congruence over process terms)*** *The relation* $\equiv \subseteq \mathcal{P} \times \mathcal{P}$ *is defined to be the smallest congruence over* $\mathcal{P}$ *which satisfies the following axioms: 1) if* $P \equiv_\alpha P'$ *then* $P \equiv P'$*; 2)* $\nu x.\surd \equiv \surd$*; 3)* $\nu x.\nu y.P \equiv \nu y.\nu x.P$*; 4)* $(\mathcal{P}, \|, \surd)$ *is an abelian monoid; 5) if* $x \notin fn(P)$ *then* $P \parallel \nu x.Q \equiv \nu x.(P \parallel Q)$*; and 6) if* $A(x_1, ..., x_n) \stackrel{def}{=} P$ *then* $A(y_1, ..., y_n) \equiv P\{^{y_1}/_{x_1}, ..., ^{y_n}/_{x_n}\}$.

A *timed labelled transition system* or *TLTS*, is a transition system in which we distinguish between transitions due to actions and evolution (passage of time). Formally, a TLTS is a tuple $(\mathcal{S}, \mathcal{L}, \rightarrow, \rightsquigarrow)$ where $\mathcal{S}$ is a set of states, $\mathcal{L}$ is a set of *labels*, $\rightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is a *transition relation* and $\rightsquigarrow \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$ is an *evolution relation.* A *rooted* TLTS $(\mathcal{S}, s_0, \mathcal{L}, \rightarrow, \rightsquigarrow)$ is a TLTS with a distinguished *initial state* s_0. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$ and $s \stackrel{d}{\rightsquigarrow} s'$ for $(s, d, s') \in \rightsquigarrow$. We write $s \xrightarrow{a}$ to mean that $\exists s' \in \mathcal{S}.\, s \xrightarrow{a} s'$.

Definition 3. ***(Process transitions and evolution)*** *The meaning of a* π_{klt} *term* P_0 *is a rooted TLTS* $(\mathcal{P}, P_0, \mathcal{A}, \rightarrow, \rightsquigarrow)$ *where* $\mathcal{A}$ *is the set of action labels described below and the relations* $\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ *and* $\rightsquigarrow \subseteq \mathcal{P} \times \mathbb{R}_0^+ \times \mathcal{P}$ *are the smallest relations satisfying the inference rules in Table 1. The elements of* $\mathcal{A}$ *are actions of the form* τ *(silent action),* $x?u$ *(reception),* $x!u$ *(trigger), or* $x!\nu u$ *(bound trigger) where* u *is a value. We let* α *range over* $\mathcal{A}$*. We write* $\text{bn}(\alpha)$ *for*

[3] We do not need a name environment, as all expressions will be closed, since the appropriate substitutions of free names are performed before evaluation takes place.

[4] $\sigma[^V/_x](x) \stackrel{def}{=} V$ and $\sigma[^V/_x](y) \stackrel{def}{=} \sigma(y)$ if $x \neq y$.

Table 1. Process transitions and evolution

$$(\text{TRIG})\ x!E \xrightarrow{x!eval(E)} \surd \qquad (\text{DEL})\ \text{if } eval(E)=0 \text{ then } \Delta E.P \xrightarrow{\tau} P$$

$$(\text{CH})\ \text{if } \sigma = match(F_i, v, \emptyset) \neq \bot \text{ then } \textstyle\sum_{i\in I} x_i?F_i@y_i.P_i \xrightarrow{x_i?v} P_i\sigma[{}^0/_{y_i}]$$

$$(\text{NEW})\ \text{if } P \xrightarrow{\alpha} P' \text{ and } x \notin \text{n}(\alpha) \text{ then } \nu x.P \xrightarrow{\alpha} \nu x.P'$$

$$(\text{PAR})\ \text{if } P \xrightarrow{\alpha} P' \text{ and } \text{bn}(\alpha) \cap \text{fn}(Q) = \emptyset \text{ then } P \parallel Q \xrightarrow{\alpha} P' \parallel Q$$

$$(\text{COMM})\ \text{if } P \xrightarrow{x!v} P' \text{ and } Q \xrightarrow{x?v} Q' \text{ then } P \parallel Q \xrightarrow{\tau} P' \parallel Q'$$

$$(\text{OPEN})\ \text{if } P \xrightarrow{x!u} P' \text{ and } x \notin \text{n}(u) \text{ then } \nu\tilde{u}.P \xrightarrow{x!\nu u} P' \text{ with } \tilde{u} = \text{n}(u)$$

$$(\text{CLOSE})\ \text{if } P \xrightarrow{x!\nu u} P' \text{ and } Q \xrightarrow{x?v} Q' \text{ then } P \parallel Q \xrightarrow{\tau} \nu\tilde{u}.(P' \parallel Q') \text{ with } \tilde{u} = \text{n}(u)$$

$$(\text{CNGR})\ \text{if } P \xrightarrow{\alpha} P',\ P \equiv Q \text{ and } P' \equiv Q' \text{ then } Q \xrightarrow{\alpha} Q'$$

$$(\text{TIDLE})\ \surd \overset{d}{\rightsquigarrow} \surd \qquad (\text{TTRIG})\ x!E \overset{d}{\rightsquigarrow} x!E$$

$$(\text{TCH})\ \textstyle\sum_{i\in I} x_i?F_i@y_i.P_i \overset{d}{\rightsquigarrow} \sum_{i\in I} x_i?F_i@y_i.P_i\{{}^{y_i+d}/_{y_i}\}$$

$$(\text{TNEW})\ \text{if } P \overset{d}{\rightsquigarrow} P' \text{ then } \nu x.P \overset{d}{\rightsquigarrow} \nu x.P'$$

$$(\text{TDEL})\ \text{if } 0 \leqslant d \leqslant eval(E) \text{ then } \Delta E.P \overset{d}{\rightsquigarrow} \Delta(E-d).P$$

$$(\text{TPAR})\ \text{if } P \overset{d}{\rightsquigarrow} P' \text{ and } Q \overset{d}{\rightsquigarrow} Q' \text{ then } P \parallel Q \overset{d}{\rightsquigarrow} P' \parallel Q'$$

$$(\text{TCNGR})\ \text{if } P \overset{d}{\rightsquigarrow} P',\ P \equiv Q \text{ and } P' \equiv Q' \text{ then } Q \overset{d}{\rightsquigarrow} Q'$$

the set of bound names of the action α, *namely* $\text{bn}(x?u) = \text{bn}(x!\nu u) \overset{def}{=} \text{n}(u)$, $\text{bn}(x!u) = \text{bn}(\tau) \overset{def}{=} \emptyset$. *We impose an additional constraint on the TLTS to guarantee maximal progress (urgency of internal actions):*

$$\textit{if } P \xrightarrow{\tau} \textit{ then } P \overset{d}{\not\rightsquigarrow} \textit{ for all } d > 0$$

The rules (CH) and (TCH) in Table 1 are of particular interest. The rule (TCH) states that a listener can let time pass by, incrementing the variables y_i according to the elapsed time. The rule (CH) states that when an event x_i is triggered with a value that matches the corresponding pattern, the process P_i is selected with the appropriate bindings, in particular y_i is bound to 0 as any waiting time has already been taken into account and added by previous applications of the (TCH) rule.

Example. Let us revisit the server response time example. Consider the execution of the device $D(b)$ when it receives a request to test the first server with links q_1 and a_1. In the given example this server takes 3.2 seconds to respond. Then the testing device will have the following execution (in this example we explicitly expand the timeout construct):

$$\begin{array}{lll}
D(b) & \xrightarrow{b?\langle 5,q_1,a_1,r_1,m_1\rangle} & q_1!.(a_1?@e.r_1!e.D(b)) \overset{5}{\triangleright} m_1!.D(b) \\
& \xrightarrow{q_1!\varnothing} & (a_1?@e.r_1!e.D(b)) \overset{5}{\triangleright} m_1!.D(b) \\
& \equiv & \nu s.((a_1?@e.r_1!e.D(b) + s?.m_1!.D(b)) \parallel \Delta 5.s!) \\
& \overset{3.2}{\rightsquigarrow} & \nu s.((a_1?@e.r_1!(e+3.2).D(b) + s?.m_1!.D(b)) \parallel \Delta(5-3.2).s!) \\
& \xrightarrow{a_1?\varnothing} & \nu s.(r_1!(0+3.2).D(b) \parallel \Delta(5-3.2).s!) \\
& \xrightarrow{r_1!3.2} & \nu s.(D(b) \parallel \Delta(5-3.2).s!) \\
& \overset{1.8}{\rightsquigarrow} & \nu s.(D(b) \parallel \Delta 0.s!) \\
& \xrightarrow{\tau} & \nu s.(D(b) \parallel s!) \equiv D(b)
\end{array}$$

3 Timed Equivalence

As explained in the introduction, all behaviours that violate hard response-time constraints of real-time systems are considered failures. Therefore we can weaken our comparison criteria to behaviours up to a given deadline. Consider the following processes: $A_1 \stackrel{def}{=} (a?.P) \stackrel{3}{\triangleright} Q$ and $A_2 \stackrel{def}{=} (a?.P) \stackrel{5}{\triangleright} Q$. Before time 3, both A_1 and A_2 have exactly the same transitions ($a?$) and evolutions. If we have a hard constraint requiring interaction before 3 time units, we don't care about their behaviour beyond time 3, and so it makes sense to identify the two processes up-to time 3. Nevertheless, these systems cannot be identified under standard notions of bisimilarity. To see this, recall the definition of timeout. We can see that A_1 has the following execution: $A_1 \stackrel{3}{\rightsquigarrow} (a?.P) \stackrel{0}{\triangleright} Q \xrightarrow{\tau} \xrightarrow{\tau} Q$ but this cannot be matched by A_2: $A_2 \stackrel{3}{\rightsquigarrow} (a?.P) \stackrel{2}{\triangleright} Q \stackrel{\tau}{\not\rightarrow}$. Hence, A_1 and A_2 cannot be identified by any of the existing timed bisimilarities for timed π-calculi, such as those in [3], [1], [6] or [2] or bisimilarities that match evolution directly (*i.e.*, $P \stackrel{d}{\rightsquigarrow} P'$ implies $Q \stackrel{d}{\rightsquigarrow} Q'$ with P' bisimilar to Q').

In [13], Schneider introduced a notion of *timed bisimilarity up-to time* T to compare the behaviour of Timed CSP processes. A good notion of observational equivalence is one which satisfies the property that whenever two processes are identified, no observer or context can distinguish between them. Such a property is satisfied by an equivalence relation which is preserved by all combinators or operators of the language, in other words, by a congruence relation (compositionality). Unfortunately Schneider's equivalence is not a congruence in the context of timed π-calculi, because, as ground bisimilarity, it is not preserved by listeners (input). In the theory of the π-calculus several alternative definitions of bisimilarity have been explored to ensure compositionality. Sangiorgi's *open bisimilarity* [12] has the desired feature: it is a congruence for all π-calculus operators. This suggests the following equivalence for dense-time π-calculi which combines Schneider's timed bisimilarity with Sangiorgi's open bisimilarity.

Definition 4. ***(Open timed-bisimulation)*** *Let $\mathcal{S}$ be a set of terms in some language equipped with a notion of substitution, where substitutions are functions $\sigma : \mathcal{S} \to \mathcal{S}$. Let $(\mathcal{S}, \mathcal{L}, \to, \rightsquigarrow)$ be a TLTS over $\mathcal{S}$. A relation $B \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$, is called an* ***open timed-simulation*** *if for all $t \in \mathbb{R}_0^+$, whenever $(P, t, Q) \in B$ then, for any substitution $\sigma : \mathcal{S} \to \mathcal{S}$ and any $d \in \mathbb{R}_0^+$ such that $d < t$:*

1. $\forall \alpha \in \mathcal{L}. \forall P' \in \mathcal{S}. P\sigma \xrightarrow{\alpha} P' \Rightarrow \exists Q' \in \mathcal{S}. Q\sigma \xrightarrow{\alpha} Q' \wedge (P', t, Q') \in B$
2. $\forall P' \in \mathcal{S}. P\sigma \stackrel{d}{\rightsquigarrow} P' \Rightarrow \exists Q' \in \mathcal{S}. Q\sigma \stackrel{d}{\rightsquigarrow} Q' \wedge (P', t-d, Q') \in B$

If B and B^{-1} are open-timed simulations, then B is called an ***open-timed-bisimulation****. Let $\Leftrightarrow \stackrel{def}{=} \{(P, u, Q) \in \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S} \mid \exists B. B$ is an open timed-bisimulation $\& \, (P, u, Q) \in B\}$. For any given $t \in \mathbb{R}_0^+$, let $\Leftrightarrow_t \stackrel{def}{=} \{(P, Q) \in \mathcal{S} \times \mathcal{S} \mid (P, t, Q) \in \Leftrightarrow\}$.*

Remark 1. For any $t \in \mathbb{R}_0^+$, $\Leftrightarrow_t$ is an equivalence relation and $\Leftrightarrow$ is the largest open timed-bisimulation.

With this definition we can now establish that $A_1 \leftrightarroweq_3 A_2$ for the processes A_1 and A_2 defined above.

Proposition 1. *For any TLTS* $M = (\mathcal{S}, \mathcal{L}, \rightarrow, \rightsquigarrow)$*, any* $t, u \in \mathbb{R}_0^+$*, and any* $P, P', P'' \in \mathcal{S}$*:*

1. *If* $P \leftrightarroweq_t P'$ *then for any* $u \leqslant t$*,* $P \leftrightarroweq_u P'$*; and*
2. *if* $P \leftrightarroweq_t P'$ *and* $P' \leftrightarroweq_u P''$ *then* $P \leftrightarroweq_{\min\{t,u\}} P''$*.*

Timed compositionality. Now we focus on the compositionality properties of open timed-bisimilarity. First, we have that it is closed under substitutions:

Lemma 1. *For any substitution* σ*, and any* $t \in \mathbb{R}_0^+$*, if* $P \leftrightarroweq_t Q$ *then* $P\sigma \leftrightarroweq_t Q\sigma$*.*

As mentioned above, a good observational equivalence should be a congruence. However, as we have argued, we only care about the observable behaviour up-to some time T, which means that the equivalence must be preserved by our operators only up to that time. Hence we are after a notion of *timed-compositionality*, characterized by a *timed-congruence*, which we now formally define:

Definition 5. ***(Timed-congruence)*** *Given some set* $\mathcal{S}$*, and a ternary relation* $R \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$*, we define R's t-**projection** to be the binary relation* $R_t \stackrel{def}{=} \{(p,q) \in \mathcal{S} \times \mathcal{S} \mid (p,t,q) \in R\}$*. R is called a t-**congruence** iff* R_t *is a congruence. R is called a **timed-congruence** if it is a t-congruence for all* $t \in \mathbb{R}_0^+$*. R is called a **timed-congruence up-to** u iff it is a t-congruence for all* $0 \leqslant t \leqslant u$*.*

Open timed-bisimilarity satisfies the following stronger property which we obtain using Lemma 1:

Lemma 2. *For any* $P, P', Q, Q_1, ..., Q_n \in \mathcal{P}$*, and any* $t \in \mathbb{R}_0^+$*, if* $P \leftrightarroweq_t P'$ *then:*

1. $\Delta E.P \leftrightarroweq_{t+e} \Delta E.P'$ *where* $e = eval(E)$
2. $\nu x.P \leftrightarroweq_t \nu x.P'$
3. $P \parallel Q \leftrightarroweq_t P' \parallel Q$
4. $x?F@y.P + \sum_{i=1}^{n} \beta_i.Q_i \leftrightarroweq_t x?F@y.P' + \sum_{i=1}^{n} \beta_i.Q_i$ *where each* β_i is of the form $x_i?F_i@y_i$.

The immediate consequence, which follows from Proposition 1 and Lemma 2, is timed-compositionality:

Theorem 1. $\leftrightarroweq_t$ *is a timed-congruence up-to t and* $\leftrightarroweq$ *is a timed-congruence.*

We also obtain the following properties as a consequence of Lemma 2, which guarantees equivalence up to the least upper bound of the pairwise time-equivalences:

Corollary 1. *For any families of terms* $\{P_i \in \mathcal{P}\}_{i \in I}$ *and* $\{Q_i \in \mathcal{P}\}_{i \in I}$*, if for each* $i \in I$*,* $P_i \leftrightarroweq_{t_i} Q_i$ *then*

1. $\prod_{i \in I} P_i \leftrightarroweq_{\min\{t_i | i \in I\}} \prod_{i \in I} Q_i$
2. $\sum_{i \in I} x_i?F_i@y_i.P_i \leftrightarroweq_{\min\{t_i | i \in I\}} \sum_{i \in I} x_i?F_i@y_i.Q_i$

4 An Abstract Machine for the π_{klt}-Calculus

We now turn our attention to the executable semantics of π_{klt}.

4.1 Abstract Machine Specification

Our abstract machine is similar to Turner's abstract machine for the π-calculus [14], but unlike Turner's, we have to take evolution over real-time and pattern-matching into account. As mentioned in the introduction, the abstract machine is based on event-scheduling, which, unlike discrete-time algorithms, does not require idle iteration cycles at times when no events are scheduled. The key idea is to treat each π_{klt} term as a *simulation event* to be executed by an event-scheduler (and not to be confused with a *communication event* in the language itself). Such event scheduler forms the heart of our abstract machine.

The global queue. The event-scheduler contains a queue of simulation events (terms) to be executed, but rather than store them all in a single linear queue, we divide them into *time-slots*, *i.e.*, sequences of all simulation events to be executed at a given instant in time. Hence the global event queue is a time-ordered queue of time-slots, each of which is a queue of terms. We describe the operation of our abstract machine by showing how it evolves in these two "dimensions" of time: the "vertical dimension" which corresponds to the execution of all terms in a single time-slot, and the "horizontal dimension", which corresponds to the advance in time, *i.e.*, the progress of the global queue.

Definition 6. ***(Global queue)*** *The set $\mathcal{R}$ of* ***global queue states****, ranged over by R, is defined by the following BNF, where T ranges over the set $\mathcal{T}$ of* ***time-slots****:*

$$R ::= (t_1, T_1) \cdot (t_2, T_2) \cdot \dots \cdot (t_n, T_n) \quad | \quad \langle\rangle$$
$$T ::= P_1 :: P_2 :: \dots :: P_m \quad | \quad \epsilon$$

where each $P_i \in \mathcal{P}$, each $t_i \in \mathbb{R}_0^+$, and for each $i \geqslant 1$, $t_i < t_{i+1}$.

Event observers and the heap. Multiple processes can trigger and/or listen to the same channel. Hence we need to keep track of each of these requests in an *observer set*[5], containing *observers, i.e.,* requests to either send a message over a channel or listen to it. We also define the *heap*, which is a map associating each channel name to its observer set.

Definition 7. ***(Channel observers and heap)*** *The set of* ***channel observers****, ranged over by O,* ***observer sets****, ranged over by Q and the set $\mathcal{H}$ of* ***heaps****, ranged over by H, are defined by the following BNFs:*

$$O ::= !v \quad | \quad ?(F, y, P, t, c)$$
$$Q ::= \{O_1, O_2, \cdots, O_n\} \quad | \quad \emptyset$$
$$H ::= x_1 \mapsto Q_1, x_2 \mapsto Q_2, ..., x_m \mapsto Q_m \quad | \quad \epsilon$$

[5] Analogous to "channel queues" in Turner's terminology.

where $v \in \mathcal{V}$*,* $F \in \mathcal{F}$*,* $P \in \mathcal{P}$*, and* $t \in \mathbb{R}_0^+$*. We denote* $H\{x \mapsto Q\}$ *for the heap where the entry for* x *is updated to* Q*. We extend this notation to indexed sets:* $H\{x_i \mapsto Q_i\}_{i\in I}$ *stands for* $H\{x_1 \mapsto Q_1\}\cdots\{x_n \mapsto Q_n\}$ *for* $I = \{1, ..., n\}$*.*

Observers of the form $!v$ are *output observers*, and denote an attempt to send a value v over the given channel. Observers of the form $?(F, y, P, t, c)$ denote *input observers*, with a pattern F to match, elapsed-time variable y, body P to execute when a message arrives and which start listening at time t. This tag t is necessary in order to assign the correct elapsed-time to y once interaction occurs. Such time-stamp is not present in Turner's machine, since his is "time agnostic". The last item, c, is a tag used to identify the original π_{klt} listener, so that each branch of a listener $\sum_{i\in I} x_i?F_i@y_i.P_i$ will have an observer $?(F_i, y_i, P_i, t, c)$ sharing the same identifier c. This is also absent from Turner's machine, since he does not implement the choice operator.

Unlike Turner's machine, any given non-empty observer set can contain both inputs and outputs simultaneously, because it is possible that the value sent over a channel does not match any of the patterns of the available input observers and thus the corresponding output observer must be suspended with the existing inputs on the same observer set. Hence the following auxiliary definitions will be useful to extract the relevant observers from a set.[6]

Definition 8. *Given an observer set* Q*, we denote:*

$$inputs(Q) \stackrel{def}{=} \{O \in Q \mid O \text{ is of the form } ?(F, y, P, t, c)\}$$
$$outputs(Q) \stackrel{def}{=} \{O \in Q \mid O \text{ is of the form } !v\}$$
$$patt(?(F, y, P, t, c)) \stackrel{def}{=} F \qquad tag(?(F, y, P, t, c)) \stackrel{def}{=} c \qquad val(!v) \stackrel{def}{=} v$$
$$inms(v, Q) \stackrel{def}{=} \{(O, \sigma) \mid O \in inputs(Q), \sigma = match(patt(O), v, \emptyset),\ \sigma \neq \bot\}$$
$$outms(F, Q) \stackrel{def}{=} \{(O, \sigma) \mid O \in outputs(Q), \sigma = match(F, val(O), \emptyset),\ \sigma \neq \bot\}$$

The last two functions give us the set of observers and bindings for successful matches between a value v (resp. a pattern F) and the patterns (resp. values) available as input (resp. output) observers in the set.

We also need the ability to remove input observers from all branches of a listener once a branch has been triggered. To this end, we define the following which removes all c tagged inputs from an observer set Q:

$$withdraw(Q, c) \stackrel{def}{=} \{O \in inputs(Q) \mid tag(O) \neq c\} \cup outputs(Q)$$

which we use to define the following function that removes all such inputs anywhere in the heap[7]:

$$rall(\epsilon, c) \stackrel{def}{=} \epsilon \quad \text{and}$$
$$rall((H, x \mapsto Q), c) \stackrel{def}{=} rall(H, c), x \mapsto withdraw(Q, c)$$

[6] We assume the standard set theoretical operations for observer sets: *e.g.*, $Q \cup \{O\}$ denotes the observer set that adds O to Q, $Q\backslash O$ is the set that results from removing O from Q, $O \in Q$ tests for membership, etc.

[7] This definition is inefficient since it traverses the entire heap. In practice, the input observers contain a list of pointers to the relevant heap entries to remove the alternatives efficiently.

Executing time-slots. We now describe the "vertical dimension" of time, *i.e.*, the execution of terms within one time-slot.

Definition 9. ***(Time-slot execution)*** *The behaviour of time-slots at a time* $t \in \mathbb{R}_0^+$ *is defined as the smallest relation* $\rightarrow_t \subseteq (\mathcal{H} \times \mathcal{T}) \times (\mathcal{H} \times \mathcal{T})$ *satisfying the rules below:*

(NIL) $(H, \surd :: T) \rightarrow_t (H, T)$
(RES) $(H, \nu x.P :: T) \rightarrow_t (H\{k \mapsto \emptyset\}, P\{^k/_x\} :: T)$ *with* k *fresh*
(SP_1) $(H, (P_1 \parallel P_2) :: T) \rightarrow_t (H, T :: P_1 :: P_2)$
(SP_2) $(H, (P_1 \parallel P_2) :: T) \rightarrow_t (H, T :: P_2 :: P_1)$
(OUT-F) *if* $H(x) = Q$ *and* $inms(eval(E), Q) = \emptyset$ *then*
$(H, x!E :: T) \rightarrow_t (H\{x \mapsto Q \cup \{!eval(E)\}\}, T)$
(OUT-S) *if* $H(x) = Q$ *and* $(?(F, y, P, u, c), \sigma) \in inms(eval(E), Q) \neq \emptyset$ *then*
$(H, x!E :: T) \rightarrow_t (rall(H, c), P\sigma[^{t-u}/_y] :: T)$
(INP-F) *if* $\forall i \in I.\, H(x_i) = Q_i$, $outms(F_i, Q_i) = \emptyset$ *and* c *fresh, then*
$(H, \sum_{i \in I} x_i?F_i@y_i.P_i :: T) \rightarrow_t (H\{x_i \mapsto Q_i \cup \{?(F_i, y_i, P_i, t, c)\}\}_{i \in I}, T)$
(INP-S) *if* $\exists i \in I.\, H(x_i) = Q_i$ *and* $(O, \sigma) \in outms(F_i, Q_i) \neq \emptyset$ *then*
$(H, \sum_{i \in I} x_i?F_i@y_i.P_i :: T) \rightarrow_t (H\{x_i \mapsto Q_i \backslash O\}, P_i\sigma[^0/_{y_i}] :: T)$
(INST) *if* $A(\tilde{x}) \stackrel{def}{=} P$ *then* $(H, A(\tilde{v}) :: T) \rightarrow_t (H, T :: P\{^{\tilde{v}}/_{\tilde{x}}\})$

The rule (NIL) simply ignores a terminated process. The (SP) rules break a parallel composition into its components and spawn their execution in the current time-slot in an arbitrary order. Unlike Turner's machine, this is non-deterministic. The (RES) rule allocates a new spot for the new channel, and initializes its observer set to empty.

The (OUT-F) rule describes the case when a message is sent over x and there is no matching listener in x's observer set (output failure). In this case we simply add a new trigger observer to x's observer set and continue. The (OUT-S) rule (output success) applies when there is a matching observer O with σ being the resulting bindings. In this case, we remove all observers belonging to the matching listener (those tagged with c), and then execute the body P of the observer, applying the substitution σ extended with the binding of the elapsed time variable y to the difference between the current time t and the time u when the receiver started listening. Note that since there might be more than one successful match, the choice is non-deterministic, contrasting again with Turner's machine.

The rules (INP-S) and (INP-F) are the dual of (OUT-S) and (OUT-F). In rule (INP-F), when attempting to execute a listener, if there are no matching triggers in the relevant observer sets, we simply add the appropriate observers to the corresponding observer sets. Note that the added observers are tagged with the current time t, and with the same tag c. On the other hand, in rule (INP-S), one of the branches succeeds in matching the pattern with an observer O and binding σ. In this case, we remove the output observer from the event's observer set and execute the body of the corresponding branch, applying the substitution σ and binding y_i to 0, since the listener did not have to wait.

Finally, the rule (INST) deals with process instantiations. This simply assumes the set of process definitions is available, and schedules the execution of the body of the definition by replacing its parameters by the arguments provided by the instantiation.

Note that there is no rule associated with the Δ operator. We specify its behaviour in the description of the global scheduler below.

Global event scheduler. Now we can define the behaviour of the global event scheduler. For this purpose we will assume we have a function $insort : \mathbb{R}_0^+ \times \mathcal{P} \times \mathcal{R} \rightarrow \mathcal{R}$ (formally defined in [10]) which, given a time, inserts a term in the appropriate time-slot in the global queue, preserving the order of time-slots w.r.t. their time-stamps.

Definition 10. ***(Scheduler)*** *The behaviour of the scheduler is given by the smallest relations* $\hookrightarrow_0, \hookrightarrow_1, \hookrightarrow_2 \subseteq (\mathcal{H} \times \mathcal{R}) \times (\mathcal{H} \times \mathcal{R})$ *which satisfy the rules below:*

(TS) *if* $(H, T) \rightarrow_t (H', T')$ *and* $T \neq \epsilon$ *then* $(H, (t, T) \cdot R) \hookrightarrow_0 (H', (t, T') \cdot R)$

(ADV) $(H, (t, \epsilon) \cdot R) \hookrightarrow_1 (H, R)$

(SCH) $(H, (t, \Delta E.P :: T) \cdot R) \hookrightarrow_2 (H, insort(t + eval(E), P, (t, T) \cdot R))$

The rule (TS) states that as long as there are terms in the current time-slot then they are executed. The (ADV) rule states that when the current time-slot is empty, execution moves on to the next available time-slot. Finally, the (SCH) rule describes the behaviour of the delay operator: to execute $\Delta E.P$, the value d of E is computed and P is inserted at time $t + d$ (where t is the current time). Note that P is inserted in $(t, T) \cdot R$ because the value of E may be 0. This may create a new time-slot, if there was none at time $t + d$.

Example. We illustrate the abstract machine with a sample execution. Consider the processes $Q \stackrel{def}{=} \Delta 3.2.x!1$ and $P \stackrel{def}{=} x?1@e.P_1$ where $P_1 \stackrel{def}{=} \Delta(5 - e).P_2$ for some P_2. Suppose that the current time is, for example, 7. The execution of the time-slot containing only $\nu x.(P \parallel Q)$, assuming the heap is initially empty (just to simplify notation) is:

$$
\begin{array}{lll}
 & (\epsilon, (7, \nu x.(P \parallel Q))) & \\
\hookrightarrow_0 & (k \mapsto \emptyset, (7, (P \parallel Q)\{^k/_x\})) & \text{(RES)+(TS)} \\
\equiv & (k \mapsto \emptyset, (7, (P\{^k/_x\} \parallel Q\{^k/_x\}))) & \\
\hookrightarrow_0 & (k \mapsto \emptyset, (7, P\{^k/_x\} :: Q\{^k/_x\})) & \text{(SP}_1\text{)+(TS)} \\
\hookrightarrow_0 & (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, (7, Q\{^k/_x\})) & \text{(INP-F)+(TS)} \\
\equiv & (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, (7, \Delta 3.2.k!1)) & \\
\hookrightarrow_2 & (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, insort(7 + 3.2, k!1, (7, \epsilon))) & \text{(SCH)} \\
\equiv & (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, (7, \epsilon) \cdot (10.2, k!1)) & \\
\hookrightarrow_1 & (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, (10.2, k!1)) & \text{(ADV)} \\
\hookrightarrow_0 & (k \mapsto \emptyset, (10.2, P_1\{^k/_x\}\{^{10.2-7}/_e\})) & \text{(OUT-S)+(TS)} \\
\equiv & (k \mapsto \emptyset, (10.2, \Delta(5 - 3.2).P_2\{^{3.2}/_e\}\{^k/_x\})) & \\
\hookrightarrow_2 & (k \mapsto \emptyset, (10.2, \epsilon) \cdot (12, P_2\{^{3.2}/_e\}\{^k/_x\})) & \text{(SCH)} \\
\hookrightarrow_1 & (k \mapsto \emptyset, (12, P_2\{^{3.2}/_e\}\{^k/_x\})) & \text{(ADV)}
\end{array}
$$

Here we used ($\textsc{sp}_1$) which selected the left process P to be executed first. This resulted in registering the input observer in k's observer set. If we had used ($\textsc{sp}_2$) instead, then Q would have been executed first, resulting in the scheduling happening first, but P would remain in the time-slot for time 7, and thus it would be executed before advancing in time.

4.2 Soundness

We now establish that reductions in our abstract machine correspond to valid π_{klt} executions, following the same approach from [14]. To do this, we first encode the states of our abstract machine as π_{klt} terms, in particular we need to encode the heap, its observer sets, time-slots and the global queue. We use $x \in H$ to denote that there is an entry for x in the heap H.

Definition 11. ***(Encoding the machine state)*** *The set of* ***triggers in the heap entry for*** *x and the set of* ***all triggers in the heap*** *are given by:*

$$triggers(Q,x) \stackrel{def}{=} \{x!v \mid !v \in outputs(Q)\}$$

$$alltriggers(H) \stackrel{def}{=} \bigcup_{x\in H} triggers(H(x),x)$$

The set of ***branches of a listener with tag*** *c is given by:*

$$branches(H,c,t) \stackrel{def}{=} \bigcup_{x\in H} alts(H(x),c,x,t) \text{ where}$$

$$alts(Q,c,x,t) \stackrel{def}{=} \{x?F@y.P\{^{y+(t-u)}/_y\} \mid ?(F,y,P,u,c) \in inputs(Q)\}$$

The set of ***all listeners in the heap*** *is given by:*

$$alllisteners(H,t) \stackrel{def}{=} \{\textstyle\sum branches(H,c,t) \mid c \in alltags(H)\}$$

where

$$alltags(H) \stackrel{def}{=} \bigcup_{x\in H} tags(H(x)) \text{ and}$$

$$tags(Q) \stackrel{def}{=} \{tag(O) \mid O \in inputs(Q)\}$$

The ***encoding of the heap*** *H* ***at time*** *t, is given by:*

$$[\![H]\!]_t \stackrel{def}{=} (\textstyle\prod alllisteners(H,t)) \parallel (\prod alltriggers(H))$$

The ***encoding of the time-slot*** *$T = P_1 :: P_2 :: \cdots :: P_n$ is:*

$$[\![T]\!] \stackrel{def}{=} P_1 \parallel P_2 \parallel \cdots \parallel P_n$$

The ***encoding of a heap/time-slot pair at time*** *t is:*

$$[\![(H,T)]\!]_t \stackrel{def}{=} [\![H]\!]_t \parallel [\![T]\!]$$

The ***encoding of the global queue*** *$R = (t_1,T_1)\cdot(t_2,T_2)\cdot ...$ is given by:*

$$[\![R]\!] \stackrel{def}{=} [\![T_1]\!] \parallel \Delta(t_2-t_1).[\![T_2]\!] \parallel \cdots \parallel \Delta(t_n-t_1).[\![T_n]\!]$$

The ***encoding of the machine state*** *(H,R) is defined as:*

$$[\![(H,R)]\!] \stackrel{def}{=} \nu x_1,...,x_k.([\![H]\!]_{curtime(R)} \parallel [\![R]\!])$$

where $\{x_1,...,x_k\}$ are the names of entries in the heap H and $curtime((t,T)\cdot R) \stackrel{def}{=} t$ is the time-stamp of the first time-slot.

Note that to create a single listener we have to quantify over the possible entries in the heap. This is because each branch of a listener may listen to different channels, and therefore, the corresponding input observers may be dispersed over multiple heap entries. The use of tag c allows us to identify all input observers

which belong to the same listener. Also, note that the definition *alts* which gives us an alternative branch of a listener, substitutes $y + (t - u)$ for y in P, where t is the current time and u is the time-stamp of the input observer, *i.e.*, when the listener began listening. This is because the encoding corresponds to taking a "snapshot" of the machine's state at time t, but each listener may have registered at some time $u \leqslant t$ so we have to take the already elapsed time into account. The encoding $[\![R]\!]$ for the global queue considers the first time-slot to represent the current one, so all future time-slots are delayed relative to the current time t_1.

Now we establish our result (we write $\Longrightarrow$ for $(\xrightarrow{\tau}) \equiv \cup \equiv$).

Lemma 3. *If* $(H, T) \rightarrow_t (H', T')$ *then* $[\![(H, T)]\!]_t \Longrightarrow [\![(H', T')]\!]_t$

Theorem 2. ***(Abstract machine soundness)***

1. *If* $(H, R) \hookrightarrow_0 (H', R')$ *then* $[\![(H, R)]\!] \Longrightarrow [\![(H', R')]\!]$
2. *If* $(H, R) \hookrightarrow_1 (H', R')$ *then* $[\![(H, R)]\!] \overset{d}{\leadsto} [\![(H', R')]\!]$ with $d = t_2 - t_1$ where $R = (t_1, T_1) \cdot (t_2, T_2) \cdot ...$
3. *If* $(H, R) \hookrightarrow_2 (H', R')$ *then* $[\![(H, R)]\!] \equiv [\![(H', R')]\!]$

5 Conclusions

We have introduced the π_{klt}-calculus, a timed extension to the asynchronous π-calculus which adds some high-level features such as pattern-matching. We have given an operational semantics in terms of timed-labelled transition systems, and developed a basic theory of time-bounded congruence. We developed an abstract machine for the calculus and established its soundness with respect to the operational semantics. To the best of our knowledge this is the first use of event-scheduling (as used in simulation) as a language interpreter. We have implemented the π_{klt}-calculus in a language called kiltera (available at `http://www.kiltera.org`) which is based on the described abstract machine. Moreover, it extends π_{klt} with primitives for distributed computing, allowing processes to be sent to remote sites and have site-dependent behaviour.

Aside from differences in the particular choice of operators, the most closely related work, to the best of our knowledge, is found in [3], [1], [6] and [2]. As mentioned before the first three consider only discrete-time variants of the π-calculus and all deal with stringent notions of timed equivalence which do not take into account time bounds. Furthermore, the equivalences in [3] are not shown to be congruences. In fact, we suspect that these equivalences are not congruences, for the same reasons that strong, ground bisimilarity is not a congruence in the π-calculus, or (non-open) timed-bisimilarity is not a congruence in π_{klt}: they are insensitive to values over channels. The other papers deal with congruences, but they are sensitive to behaviours beyond time-bounds, distinguishing processes that should be identified when considering hard constraints.

In terms of execution, to the best of our knowledge, there is no other abstract machine for a timed π-calculus, and the only other implementation is that of the $TD\pi$-calculus [4]. This implementation is quite different from ours, both in terms

of the execution model and architecture. Firstly, it uses of a clock-tick model rather than event-scheduling. Secondly, instead of using an abstract machine, it translates the source language (TiMo) to Java code, and thus depends on the Java run-time system and additional libraries. There are further differences regarding distributed execution, but these fall outside the scope of this paper. For a more detailed comparison with this and other related work, we refer the reader to [10].

There are several possible future lines of research including the development of a type system, weaker notions of equivalence and refinement relations with appropriate axiomatizations, as well as symbolic methods to help analyze systems. The mentioned extension of π_{klt} and kiltera to distribution will be described in a future paper.

References

1. Berger, M.: Basic Theory of Reduction Congruence for Two Timed Asynchronous π-Calculi. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 115–130. Springer, Heidelberg (2004)
2. Chen, J.: A proof system for weak congruence in timed π-calculus. Tech. Report 2004-13, LIFO, Université d'Orléans (2004)
3. Ciobanu, G.: Behaviour Equivalences in Timed Distributed π-calculus. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) Soft-Ware Intensive Systems. LNCS, vol. 5380, pp. 190–208. Springer, Heidelberg (2008)
4. Ciobanu, G., Juravle, C.: MCTools: A Software Platform for Mobility and Timed Interaction. Tech. Report FML-09-01, Formal Methods Laboratory – Romanian Academy – Iasi Branch (February 2009)
5. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
6. Kuwabara, H., Yuen, S., Agusa, K.: Congruence Properties for a Timed Extension of the π-Calculus. In: Proc. of DSN 2005 Workshop: Dependable Software, Tools and Methods, pp. 207–214 (2005)
7. Lee, J.Y., Zic, J.: On modeling real-time mobile processes. In: Proc. of ACSC 2002, January 2002, pp. 139–147 (2002)
8. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and ECS-LFCS-89-86 86, Computer Science Dept., University of Edinburgh (March 1989)
9. Posse, E.: Modelling and Simulation of dynamic structure, discrete-event systems. Ph.d. thesis, School of Computer Science. McGill University (August 2008)
10. Posse, E.: A real-time extension to the π-calculus. Tech. Report 2009-557, School of Computing – Queen's University (2009), http://www.cs.queensu.ca
11. Prisacariu, C., Ciobanu, G.: Timed Distributed π-Calculus. Tech. Report FML-05-01, Institute of Computer Science, Romanian Academy (2005)
12. Sangiorgi, D.: A theory of bisimulation for the π-calculus. Tech. Report ECS-LFCS-93-270, University of Edinburgh (1993)
13. Schneider, S.: An operational semantics for Timed CSP. Information and Computation (1995)
14. Turner, D.N.: The polymorphic Pi-calculus: Theory and Implementation. Ph.d. thesis, Univ. of Edinburgh (1996)
15. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of modeling and simulation, 2nd edn. Academic Press, London (2000)

Fuzzy-Timed Automata*

F. Javier Crespo, Alberto de la Encina, and Luis Llana

DSIC, Universidad Complutense de Madrid, Spain
javier.crespo@fdi.ucm.es, {albertoe,llana}@sip.ucm.es

Abstract. Timed automata theory is well developed in literature. This theory provides a formal framework to model and test real-time systems. This formal framework supplies a way to describe transitions among states with timing constrains. These constraints are usually expressed with logic formulas involving the system clocks. The time domain of these clocks usually is considered dense, that is, the clocks take values in the real or rational numbers. Dealing with a domain like this can be hard, specially if we consider end points of intervals.

In this paper, we present a modification of the model that allows to use real time in an easier, more powerful and reliable approach for computing systems. Our proposed model exploits the concepts of fuzzy set theory and related mathematical frameworks to get a more flexible approach.

Keywords: Conformance Testing, Timed Automata, Fuzzy Set Theory.

1 Introduction

Over the last decades Formal Methods have attracted the attention of researches all over the world. One of the first, and also one of the most important, enhancements to formal methods was the inclusion of temporal features. Just from the beginning, one the of the most important issues was the nature of time: whether the time is a discrete domain [8,16,15] or a dense one [19,3,5]. The authors in favor of a dense time domain argued that its expressive power is greater than the one of a discrete time domain. Those in favor of a discrete time domain argued that it is more realistic since, for instance, you can't measure $\sqrt{2}$ seconds.

As aforementioned, time can be considered discrete. In this case it is composed of sequential instants. Mathematically speaking a discrete time domain has a order relation that is isomorphic to the order relation of the natural numbers. This model implies the existence of abrupt jumps. Therefore, even the absence of vagueness, a discrete time model becomes imprecise in the real world.

On the other hand, time can be modeled to be dense. A dense time domain has an order relation similar to the one of the real numbers or the rational numbers: between two time instants there is always another time instant. In others words, there are not any abrupt jumps. When working with a dense time domain it is

* Research partially supported by the Spanish MCYT projects TIN2006-15578-C02-01 and TIN2009-14312-C02-01.

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 140–154, 2010.
© IFIP International Federation for Information Processing 2010

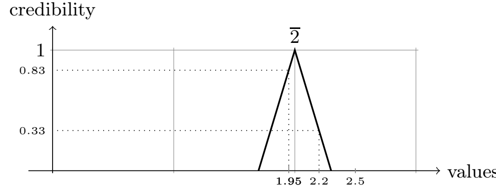


Fig. 1. Fuzzy number 2

usually necessary to discretize it. For instance, in Lazy Hybrid Automata [1], they consider that infinite precision is not possible so they discretize the continuous values by considering intervals.

Neither the discrete nor the dense models of time are capable to model properly the concept of time that humans beings have which it is inherently vague. In this paper, we propose the use of a structure that wraps the concept itself assuming the uncertainty and vagueness. In this context vagueness is not necessarily a criticism, but just a fact. The most popular approaches to handling vagueness and uncertainty as partial ignorance are Bayes theory, Shafer's evidence theory, the transferable belief model, and the possibility theory which are completely related to fuzzy sets.

Fuzzy set theory [20,21,14] provides a formal framework for the representation of vagueness. Our perception of reality is not perfect, although things can be *true* or *false*; our *environment* can make us doubt about a truth assessment. All measurement devices have an intrinsic error: if a thermometer indicates that the temperature is 35.4°C, we know that the actual temperature is around that measurement. Something similar happens with time. We can claim that *it takes us an hour to go to work*; if the trip to work lasts 57 minutes in a particular day we know that the trip has lasted as usual, otherwise if it takes us 81 minutes we know that the trip has not lasted as usual. We can also be more accurate and give a degree of *confidence* of the measurement, which is a number in the interval $[0, 1]$ with being 1 the maximum degree of confidence and 0 the minimum. In this way, we can assess that 57 minutes is equal to 1 hour with a confidence degree close to 1, while 81 minutes has a confidence degree close to 0.

Therefore, a fuzzy number can be seen as a mapping from the set of real numbers to the interval $[0, 1]$. In Figure 1 we have depicted the fuzzy number 2, denoted by $\overline{2}$. In the figure we can observe that 1.95 is relatively close to 2 so it has a high confidence level 0.83. On the contrary, 2.2 is further from 2 so it has a lower confidence level 0.3, and 2.5, that is even further, has a confidence level of 0.

Timed automata theory is well developed in literature [2,3,7,17]. This theory provides a formal framework to model and test real-time systems. This formal framework supplies a way to describe transitions among states with timing constraints. The time model usually adopted in this theory is a dense one. Nevertheless, the numbers appearing in the time constraints they introduced always range over the set of natural numbers $\mathbb{N}$. Hence, what they are really doing is a

discretization of time. As aforementioned, we do not think this is the best way to model time. Hence, we adapt this framework to include fuzzy time constraints.

Once we have defined the fuzzy-timed specifications, it is necessary to check if the implementations meet them. One of the basic relationships between specifications and implementations is trace equivalence. But in this point there is a problem, let us suppose that a specification requires that an action a must be executed in the time interval $[1, 3]$. On the one hand, since we are in a fuzzy environment, we allow an implementation to execute action a at instant 3.0001; but, on the other hand, we cannot consider incorrect an implementation that *always* executes the action a within $[1, 3]$. This could be solved if the implementation relation were the trace inclusion. However, this relation is not completely satisfactory because an implementation that does not make any action at all is considered correct. Therefore, the implementation relation should be trace equivalence for the interval $[1, 3]$ and, at the same time, it should have some tolerance outside that interval.

Another important aspect of a theory is having the proper software tools. We have not developed any tool yet, but we have expressed our fuzzy implementation relations in terms of ordinary timed automata. In this way we can use the well known tools like Uppaal [4].

The relationship between fuzzy set theory and automata theory is not new. Fuzzy automata have been used to deal with different science fields: imprecise specifications [11], modeling Learning Systems [18] and many others. Fuzziness has been introduced in the different components of the automata: states, transitions, and actions [18,13,6]. A similar work of ours has been made in [7]. They use a many-valued logic in the transitions of the automata, and although they do not use fuzzy logic, their approach logic is similar to ours.

Probabilistic and stochastic models [12,9] might be considered related to the fuzzy model presented in this paper. In these models, the time when an action is performed follows a given random variable. Before setting a probability or a random variable in a model, a thorough statistical analysis should be performed. Unfortunately this requirement is difficult, if not impossible, to achieve. Thus, the specifier must choose the probability or a random variable based on her own experience. This experience fits better in a fuzzy environment.

The rest of the paper is organized as follows. Next, in Section 2 we introduce some concepts of fuzzy logic that are used along the paper. After that, in Section 3 we introduce the concept of fuzzy specifications, implementations and fuzzy time conformance. In Section 4 we present a case study to show the main characteristics of our model. Next, in Section 5 we show how to compute our fuzzy relations in terms of ordinary timed automata. Finally in Section 6 we give some conclusions and future work guidelines.

2 Preliminaries

In this paper we do not assume that the reader is familiar with fuzzy logic concepts. Therefore, we present some basic concepts of fuzzy logic.

2.1 Fuzzy Relations

In ordinary logic, a set or a relation is determined by its characteristic function: a function that returns true if the element is in the set (or if some elements a related) and false otherwise. In the fuzzy framework we do not have that clear distinction between truth and falsehood; instead we have a complete range of values in the interval $[0, 1]$; the larger is the value, the more confidence we have in the assessment. In this paper we consider relations of real numbers $\mathbb{R}$. So a fuzzy relation is a mapping from the Cartesian product $\mathbb{R}^n$ into the interval $[0, 1]$

Definition 1. *A* fuzzy relation $\overline{A}$ *is a function* $\overline{A} : \mathbb{R}^n \mapsto [0, 1]$*. Let* $x \in \mathbb{R}^n$*, we say that* x *is* not included *in* $\overline{A}$ *if* $\overline{A} = 0$*; we say that* x *is* fully included *in* $\overline{A}$ *if* $\overline{A} = 1$*. The* kernel *of* $\overline{A}$ *is the set of elements that are fully included in* $\overline{A}$ □

The notion of α-cut is very important in fuzzy logic. Intuitively it establishes a credibility threshold. If we have a fuzzy relation $\overline{A}$, we accept that the relation is true for x if $\overline{A}(\overline{x})$ is above that threshold.

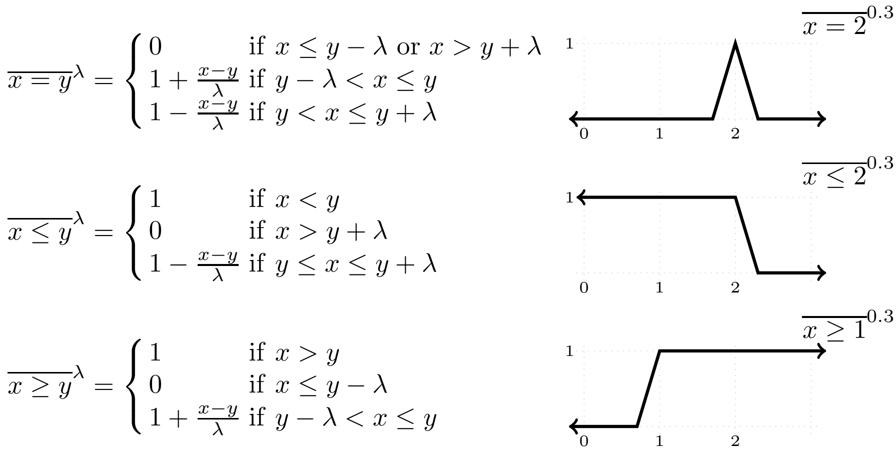
Definition 2. *Let* $\overline{A} : \mathbb{R}^n \mapsto [0, 1]$ *be a fuzzy relation and* $\alpha \in [0, 1]$*. We define the* α-cut of $\overline{A}$*, written* $\mathsf{cut}_\alpha(\overline{A})$*, as* $\mathsf{cut}_\alpha(\overline{A}) = \{x \in \mathbb{R}^n \mid \overline{A}(x) \geq \alpha\}$*.* □

Example 1. In this paper we consider the following fuzzy relations. Let us consider a non negative real number $\lambda \geq 0$,

$$\overline{x = y}^{\lambda} = \begin{cases} 0 & \text{if } x \leq y - \lambda \text{ or } x > y + \lambda \\ 1 + \frac{x-y}{\lambda} & \text{if } y - \lambda < x \leq y \\ 1 - \frac{x-y}{\lambda} & \text{if } y < x \leq y + \lambda \end{cases}$$

$$\overline{x \leq y}^{\lambda} = \begin{cases} 1 & \text{if } x < y \\ 0 & \text{if } x > y + \lambda \\ 1 - \frac{x-y}{\lambda} & \text{if } y \leq x \leq y + \lambda \end{cases}$$

$$\overline{x \geq y}^{\lambda} = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{if } x \leq y - \lambda \\ 1 + \frac{x-y}{\lambda} & \text{if } y - \lambda < x \leq y \end{cases}$$



Note that these fuzzy relations are not order relations. We use these functions to describe fuzzy-timed automata that we present in Section 3. The only property we use, which simplifies the writing, is the commutativity of the equality $\overline{x = y}^{\lambda} = \overline{y = x}^{\lambda}$, viewed as binary operation over $\mathbb{R}$. □

2.2 Triangular Norms

A triangular norm (abbreviated t-norm) is a binary operation used in fuzzy logic to generalize the *conjunction* in propositional logic. In order to be able to generalize the conjunction, we have to analyze its basic properties: commutativity

$p \wedge q = q \wedge p$, associativity $(p \wedge q) \wedge r = p \wedge (q \wedge r)$, identity true $\wedge\, p = p$, and nilpotency false $\wedge\, p =$ false.

Therefore, we require a t-norm to satisfy similar properties. We also require an extra property: monotonicity. Intuitively, the resulting truth value does not decrease if the truth values of the arguments increase.

Definition 3. *A* t-norm *is a function* $T : [0,1] \times [0,1] \mapsto [0,1]$ *which satisfies the following properties:*

- *Commutativity:* $T(x, y) = T(y, x)$.
- *Monotonicity:* $T(x, y) \leq T(z, u)$ *if* $x \leq z$ *and* $y \leq u$.
- *Associativity:* $T(x, T(y, z)) = T(T(x, y), z)$.
- *Number 1 is the identity element:* $T(x, 1) = x$.
- *Number 0 is nilpotent:* $T(x, 0) = 0$[1].

□

Since t-norms are associative, we can generalize them to lists of numbers:

$$T(x_1, x_2, \ldots, x_{n-1}, x_n) = T(x_1, T(x_2, \ldots, T(x_{n-1}, x_n) \ldots))$$

Example 2. The following t-norms are often used:

Łukasiewicz t-norm: $T(x, y) = \max(0, x + y - 1)$. We represent this t-norm with the symbol $\curlywedge$.

Gödel t-norm: $T(x, y) = \min(x, y)$. We represent this t-norm with the symbol $\overline{\wedge}$.

Product t-norm: $T(x, y) = x \cdot y$ (real number multiplication). We represent this t-norm with the symbol $\bigstar$.

We assume that any t-norm used has a symbol, we use the symbol $\triangle$ to represent a generic t-norm. We denote by $[\![\triangle]\!]$ the function associated with the symbol $\triangle$. For instance $[\![\overline{\wedge}]\!]$ is the min function. □

3 Fuzzy-Timed Automata

In this Section we introduce the basic notions of fuzzy-timed automata. But first, in order to make the paper self-contained, we first recall some common definitions of timed automata. Later we use and adapt these definitions to cope with fuzzy-timed automata. Apart from using these definitions later, we use ordinary timed automata to represent implementations of the fuzzy specifications.

Definition 4. Actions. *We assume that we have a finite alphabet of actions. The set of actions is denoted by* Acts. *Actions are ranged over by* $a, b, c, \ldots$

Clocks. *A clock is a real valued variable. Clocks are ranged over by* $x, y, z, \ldots$ *We assume that we have a finite set of clocks denoted by* Clocks

Clock valuations. *A clock valuation* $u : \mathsf{Clocks} \mapsto \mathbb{R}^+$ *assigns non-negative real values to the clocks. The valuations of the clocks are denoted by* $u, v, \ldots$ *We denote by* 0 *the clock valuation where all clocks are set to* 0.
We use the following notation for valuations.

[1] As a matter of fact, this property can be easily deduced from the others.

– *Let u be a valuation and let $d \in \mathbb{R}^+$. The valuation $u + d$ denotes the valuation $(u + d)(x) = u(x) + d$.*
– *Let u be a valuation and r a set of clocks. We denote the reset of all clocks of r in u, written as $u[r]$, as a valuation that maps all clocks in r to 0 and leaves invariant the rest of clocks.*

Clock constraints. *A clock constraint is a formula consisting of conjunctions of atomic relations of the form $x \bowtie n$ or $x-y \bowtie n$ where $n \in \mathbb{N}$, $x, y \in \mathsf{Clocks}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$. We denote the set of clock constraints by $\mathcal{C}$.*
Let u be a valuation and $C \in \mathcal{C}$, we write $u \vDash C$ when the valuation u makes the clock constraint C true.

Timed Automata. *A timed automaton is a tuple (L, l_0, E, I) where:*
– *L is a finite set of locations.*
– *$l_0 \in S$ is the initial location.*
– *$E \subseteq L \times \mathsf{Acts} \times \mathcal{C} \times 2^{\mathsf{Clocks}} \times L$ is the set of edges; we write $l \xrightarrow{a,C,r} l'$ whenever $(l, a, C, r, l') \in E$.*
– *$I : L \mapsto \mathcal{C}$ is a function that assigns invariants to locations. As it is usual, the invariants of locations consist of conjunctions of atoms of the form $x \leq n$ where $n \in \mathbb{N}$.*

Operational semantics. *The operational semantics of a timed automaton is a timed labeled transition system whose states are pairs (l, u) where l is a location, u is a valuation of clocks, and its transitions are:*
– *$(l, u) \xrightarrow{d} (l, u + d)$ if $d \in \mathbb{R}^+$ and $u + d \vDash I(l)$.*
– *$(l, u) \xrightarrow{a} (l', u[r])$ if $l \xrightarrow{a,C,r} l'$ and $u \vDash C$ and $u[r] \vDash I(l')$.*

Automata traces. *Let $A = (L, l_0, E, I)$ be an automaton. A trace is a sequence $t = (d_1, a_1)(d_2, a_2) \ldots (d_n, a_n) \in (\mathbb{R}^+ \times \mathsf{Acts})^*$, written as $t \in \mathsf{tr}(A)$, if there is a sequence of transitions*

$$(l_0, 0) \xrightarrow{d_1}\xrightarrow{a_1} (l_1, u_1) \xrightarrow{d_2}\xrightarrow{a_2} (l_2, u_2) \cdots \xrightarrow{d_n}\xrightarrow{a_n} (l_n, u_n)$$

Conformance relation. *Among the different conformance notions in the timed automata literature we want to recall the trace inclusion and trace equivalence relations: $A_1 \leq_{tr} A_2 \Leftrightarrow \mathsf{tr}(A_1) \subseteq \mathsf{tr}(A_2)$ and $A_1 \equiv_{tr} A_2 \Leftrightarrow \mathsf{tr}(A_1) = \mathsf{tr}(A_2)$* $\square$

As we have indicated, our objective is to define *fuzzy-timed* automata. In timed automata theory, time is expressed in the time constraints. Hence, we need to modify these constraints in order to be able to introduce fuzziness. In ordinary timed automata theory, the time constraints consist of conjunctions of inequalities. Instead of the ordinary crisp inequalities in Definition 4, we use their fuzzy counterparts appearing in Example 1. We could have more freedom in allowing general convex fuzzy sets, but we have preferred to keep our constraints close to the original ones so we can use the theory developed for timed automata.

The role of a conjunction in Fuzzy Theory is played by t-norms. There is not a *canonical* t-norm, in Example 2 we have 3 of the more used t-norms. We have preferred to allow any of the available t-norms.

The time constraints be divided in two groups: The general fuzzy constraints that appear as the constraints attached to the actions; and the set of restricted

constraints attached to location invariants where only inequalities of the form $\overline{x \leq n}^{\lambda}$ are allowed.

Definition 5

1. *A* fuzzy constraint *is a formula built from the following B.N.F.:*

$$C ::= \mathsf{True} \mid C_1 \vartriangle C_2 \mid \overline{x \bowtie n}^{\lambda} \mid \overline{x - y \bowtie n}^{\lambda}$$

   where $\vartriangle$ *is a t-norm,* $\bowtie \in \{\leq, =, \geq\}$*,* $x, y \in \mathsf{Clocks}$*,* $\lambda \in \mathbb{R}^{+}$*, and* $n \in \mathbb{N}$*. We denote the set of fuzzy constraints by* $\mathcal{FC}$*.*

2. *A* fuzzy restricted constranint *is the subset of fuzzy constraints defined by the following B.N.F.:*

$$C ::= \mathsf{True} \mid C_1 \vartriangle C_2 \mid \overline{x \leq n}^{\lambda}$$

   where $\vartriangle$ *is a t-norm,* $x \in \mathsf{Clocks}$*,* $\lambda \in \mathbb{R}^{+}$*, and* $n \in \mathbb{N}$*. We denote the set of restricted fuzzy constraints by* $\mathcal{RFC}$*.* □

In timed automata theory, the constraints are used to decide if the automata can stay in a location and to decide if a transition can be executed. All this is done by checking if a valuation satisfies the corresponding constraint. In fuzzy theory the notion of satisfaction is not crisp, we do not have a boolean answer but a range in the interval $[0, 1]$. Therefore, we do not have if a constraint is true or false but a *satisfaction grade* of a constraint.

Definition 6. *Let* u *be a clock valuation and* C *be a fuzzy constraint, we inductively define the* satisfaction grade of C in u*, written* $\mu_C(u)$*, as*

$$\mu_C(u) = \begin{cases} 1 & \text{if } C = \mathsf{True} \\ \overline{u(x) \bowtie n}^{\lambda} & \text{if } C = \overline{x \bowtie n}^{\lambda},\ \bowtie \in \{\leq, =, \geq\} \\ \overline{u(x) - u(y) \leq n}^{\lambda} & \text{if } C = \overline{x - y \leq n}^{\lambda},\ \bowtie \in \{\leq, =, \geq\} \\ [\![\vartriangle]\!](\mu_{C_1}(u), \mu_{C_2}(u)) & \text{if } C = C_1 \vartriangle C_2 \end{cases}$$

Let us remark that $\mu_C(u) \in [0, 1]$*.* □

Once the time constraints are defined, the definition of the fuzzy-timed automata is quite straightforward. It consists of replacing the ordinary time constraints by fuzzy time constraints.

Definition 7. *A* fuzzy-timed *automaton is a tuple* (L, l_0, E, I) *where:*

- L *is a finite set of locations.*
- $l_0 \in S$ *is the initial location.*
- $E \subseteq L \times \mathsf{Acts} \times \mathcal{FC} \times 2^{\mathsf{Clocks}} \times L$ *is the set of edges; we write* $l \xrightarrow{a,C,r} l'$ *whenever* $(l, a, C, r, l') \in E$*.*
- $I : L \mapsto \mathcal{RFC}$ *is a function that assigns invariants to locations.*

Let us note that the clock constraints in the edges have the general form as indicated in Definition 5-1, while the location invariants have the restricted form as indicated in Definition 5-2. □

Next we are going to define the operational semantics of fuzzy-timed automata. We need it to obtain the fuzzy traces that are used for the conformance relations. This operational semantics is given in terms of transitions, which are enabled when time constraints hold. Since we do not have crisp time constraints, the transitions must be decorated with a real number $\alpha \in [0, 1]$. This number indicates its certainty.

In order to define the operational semantics we need a t-norm. Let us explain the reason. In the definition of the operational semantics of a ordinary timed automaton, the action transitions requires the condition "$u \vDash C$ and $u[r] \vDash I(l')$". This conjunction must be transformed into its fuzzy version: a t-norm.

Definition 8. *Let $fA = (L, l_0, E, I)$ be a fuzzy-timed automaton and $\triangle$ be a t-norm. The $\triangle$-*operational semantics of *fA is the labeled transition system whose states are $(l, u) \in L \times \mathsf{Clocks}$, the initial state is $(l_0, 0)$, and the transitions are:*

1. $(l, u) \xrightarrow{d}_{\alpha} (l, u + d)$ *whenever* $\mu_{I(l)}(u + d) = \alpha$
2. $(l, u) \xrightarrow{a}_{\alpha} (l', u[r])$ *when ever* $l \xrightarrow{a,C,r} l'$, $\alpha_1 = \mu_{I(l')}(u[r])$, $\alpha_2 = \mu_C(u)$ *and* $\alpha = [\![\triangle]\!](\alpha_1, \alpha_2)$. □

This operational semantics is not a pure timed transition system because the transitions are decorated with a real number $\alpha \in [0, 1]$ indicating the certainty to be executed. Anyway it is desirable that it has the main properties of timed transition systems: time determinism and time additivity.

Proposition 1. *Let $fA = (L, l_0, E, I)$ be a fuzzy-timed automaton and $\triangle$ be a t-norm, the $\triangle$-operational semantics of fA holds the following properties:*

Determinism. *If $(l, u) \xrightarrow{d}_{\alpha_1} (l_1, u_1)$ and $(l, u) \xrightarrow{d}_{\alpha_2} (l_2, u_2)$, then $\alpha_1 = \alpha_2$, $l_1 = l_2$ and $u_1 = u_2$.*

Additivity. *If $(l, u) \xrightarrow{d_1}_{\alpha_1} (l_1, u_1) \xrightarrow{d_2}_{\alpha_2} (l_2, u_2)$ then $(l, u) \xrightarrow{d_1+d_2}_{\alpha_2} (l_2, u_2)$*

Proof. To prove this it is enough to take into account that time transitions do not change location and the clock valuation is increased with the passing of time. □

As a further remark let us observe the α_2 of the transition $(l, u) \xrightarrow{d_1+d_2}_{\alpha_2} (l_2, u_2)$ in the time additivity property. In this case we do not have to combine it with α_1 because the constraints considered in the locations have the form $\overline{x \leq n}^{\lambda}$, therefore $\alpha_1 \geq \alpha_2$. Intuitively if the automaton stays in a location, the likelihood of remaining in it can only decrease.

This operational semantics allows to define the traces of a fuzzy-timed automaton. The traces of a timed automaton consist of sequences of pairs, the first element of the pair expresses how long the automaton has stayed in a location

and the second one the action that has made a location change. Now all these transitions are decorated with the certainty of being performed. So whenever the automaton stays in a location, we have an $\alpha \in [0,1]$ indicating the certainty of the automaton to stay in the location; and when a action transition is performed, we need a $\beta \in [0,1]$ indicating its certainty.

Definition 9. *Let* $fA = (L, l_0, E, I)$ *be a fuzzy-timed automaton and let* $\triangle$ *be a t-norm. We say that* $t = (d_1, \alpha_1, a_1, \beta_1)(d_2, \alpha_2, a_2, \beta_2) \dots (d_n, \alpha_n, a_n, \beta_n) \in (\mathbb{R}^+ \times [0,1] \times \mathsf{Acts} \times [0,1])^*$ *is a* $\triangle$-fuzzy trace of fA, *written* $t \in \mathsf{ftr}_\triangle(fA)$, *if there is a sequence of transitions*

$$(l_0, 0) \xrightarrow{d_1}_{\alpha_1} \bullet \xrightarrow{a_1}_{\beta_1} (l_1, u_1) \xrightarrow{d_2}_{\alpha_2} \bullet \xrightarrow{a_2}_{\beta_2} (l_2, u_2) \cdots \xrightarrow{d_n}_{\alpha_n} \bullet \xrightarrow{a_n}_{\beta_n} (l_n, u_n)$$

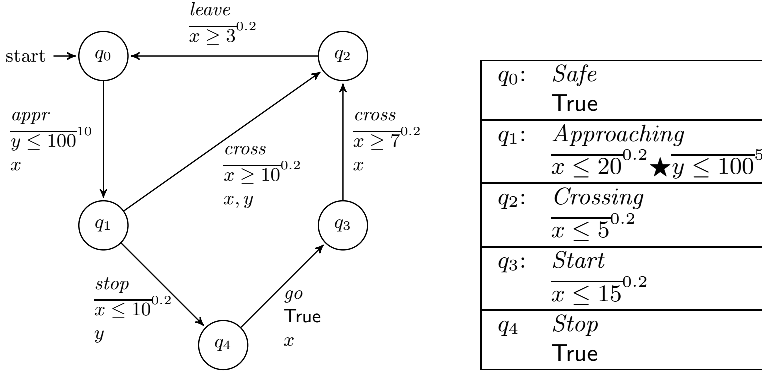
in the $\triangle$*-operational semantics of* fA*. In the previous sequence of transitions we have not specified the bullet states* $\bullet$ *to simplify the reading; they are always determined by the previous state:* $(l_i, u_i) \xrightarrow{d_{i+1}}_{\alpha_{i+1}} (l_i, u_i + d_{i+1})$ □

As aforesaid, we consider implementations are real-time systems. When we check these systems we obtain real time measurements. We need to check if those measurements meet the fuzzy constraints given by the fuzzy specification. Therefore we assume that we have traces generated from a (non-fuzzy) timed automaton. These traces are of the form $(d_1, a_1)(a_2, d_2) \cdots (d_n, a_n)$. We have to compare them to the traces of the fuzzy automaton that have the form $(d_1, \alpha_1, a_1, \beta_1)(a_2, \alpha_n, d_2, \beta_2) \dots (d_n, \alpha_n, a_n, \beta_n)$. In order to have a true/false assessment, first we have to combine the certainty values $(\alpha_1, \beta_1, \alpha_2, \beta_2, \dots, \alpha_n, \beta_n)$ with a t-norm and to compare the result to a given threshold $\alpha \in [0,1]$.

The conformance relations we want to mimic in our fuzzy framework are trace inclusion and trace equivalence. The first one is easier and corresponds to part 1 of Definition 10. Trace inclusion requires that all traces in the implementation are traces of the specification. In our case we do not have a crisp membership relation, instead we have the certainty values, a t-norm, and a threshold. Hence we compound the uncertainty values with the t-norm, we accept the trace if the obtained value is above the given threshold.

The relation corresponding to trace equivalence is in part 2 of Definition 10. It cannot be just trace equivalence. To understand the reason let us suppose that the specification allows the execution of action a in a given location with the fuzzy constraint $\overline{x \leq 3}^{0.2}$ and the threshold is 0.9. In this case we allow an implementation to execute the action a at time 3.01. But we do not want to force a correct implementation to execute that action at that time. That is, an implementation that always executes action a in the interval $[0,3]$ cannot be considered incorrect. To avoid this problem we require trace equivalence for the kernel (Definition 1) of the constraints and just trace inclusion for the rest of elements.

Definition 10. *Let* A *be a timed automaton,* fA *be a fuzzy-timed automaton,* $\triangle_1$ *and* $\triangle_2$ *be t-norms, and* $\alpha \in [0,1]$*.*



q_0:	*Safe* True
q_1:	*Approaching* $\overline{x \leq 20}^{0.2} \bigstar \overline{y \leq 100}^{5}$
q_2:	*Crossing* $\overline{x \leq 5}^{0.2}$
q_3:	*Start* $\overline{x \leq 15}^{0.2}$
q_4	*Stop* True

Fig. 2. The *train* automaton

1. A is $\triangle_1$-conformance with respect the $\triangle_2$-operational semantics of fA with an alpha cut of α, $A \ \mathsf{fconf}^{\alpha}_{\triangle_1,\triangle_2} \ fA$, *if for any trace* $(d_1, a_1)\ldots(d_n, a_n) \in \mathsf{tr}(A)$ *there exist a trace* $(d_1, \alpha_1, a_1, \beta_1)\ldots(d_n, \alpha_n, a_n, \beta_n) \in \mathsf{ftr}_{\triangle_2}(fA)$ *such that* $[\![\triangle_1]\!](\alpha_1, \beta_1, \alpha_2, \beta_2, \ldots, \alpha_n, \beta_n) \geq \alpha$.
2. A is maximally $\triangle_1$-conformance with respect the $\triangle_2$-operational semantics of fA with an alpha cut of α, *written* $A \ \mathsf{fconfm}^{\alpha}_{\triangle_1,\triangle_2} \ fA$, *if* $A \ \mathsf{fconf}^{\alpha}_{\triangle_1,\triangle_2} \ fA$ *and for any fuzzy trace* $(d_1, 1, a_1, 1)(d_2, 1, a_2, 1)\cdots(d_n, 1, a_n, 1) \in \mathsf{ftr}_{\triangle_2}(fA)$ *we have* $(d_1, a_1)(d_2, a_2)(d_n, a_n) \in \mathsf{tr}(A)$ □

4 Case study

In this section, we present a case study of an automaton with five states. This automaton, which is adapted from the Uppaal demo directory, appears in Figure 2. It represents a train passing through a crossing.

First of all let us show the need of a t-norm to define the operational semantics. Let us suppose that we measure that a train has stayed in state q_0 for 101 time units. The credibility of the corresponding transition of the fuzzy-timed automaton depends on the used t-norm. We have a valuation u where the value of the clock y is 101, so we have $\overline{y \leq 100}^{10} = 0.9$ (the constraint in the transition) and $\overline{y \leq 100}^{5} = 0.8$ (the constraint in the invariant of state q_2). So we have $(101, 1, appr, 0.8) \in \mathsf{ftr}_{\barwedge}(train)$, $(101, 1, appr, 0.72) \in \mathsf{ftr}_{\bigstar}(train)$, and $(101, 1, appr, 0.7) \in \mathsf{ftr}_{\curlywedge}(train)$ Thus, if we observe that an implementation A_1 verifies $(101, appr) \in \mathsf{tr}(A_1)$, we can say $A_1 \ \mathsf{fc\not{o}nf}^{0.8}_{\triangle,\bigstar} \ train$ and $A_1 \ \mathsf{fc\not{o}nf}^{0.8}_{\triangle,\curlywedge} \ train$, but it might $A_1 \ \mathsf{fconf}^{0.8}_{\triangle,\barwedge} \ train$.

Now let us focus on the loop among the states q_0, q_1, and q_2. Since the clock x is set to 0 in each transition, the following traces do not depend on any particular t-norm, that is, for any t-norm $\triangle$ we have

$$(20, 1, appr, 1)(10.02, 1, cross, 0.9)(4, 1, cross, 1)$$
$$(20, 1, appr, 1)(10.01, 1, cross, 0.95)(2.98, 1, cross, 0.9) \in \mathsf{ftr}_{\triangle}(train)$$

If we take the certainty values of the trace and we compound them under the different t-norms, we have

$$\begin{array}{l}[\![\bar{\wedge}]\!](1,1,1,0.9,1,1,1,1,1,0.95,1,0.9)=0.9\\ [\![\curlywedge]\!](1,1,1,0.9,1,1,1,1,1,0.95,1,0.9)=0.77\\ [\![\bigstar]\!](1,1,1,0.9,1,1,1,1,1,0.95,1,0.9)=0.76\end{array}$$

Therefore, if an implementation A_2 exhibit the trace

$$(20, appr)(10.02, cross)(4, cross)(20, appr)(10.01, cross)(2.98, cross) \in \mathsf{tr}(A_2)$$

We can say that $A_2\ \mathsf{fc\not{o}nf}^{0.9}_{\curlywedge,\triangle}\ train$ and $A_2\ \mathsf{fc\not{o}nf}^{0.9}_{\bigstar,\triangle}\ train$; but A_2 is still a candidate to conforms the fuzzy automaton with respect the Gödel t-norm, that is, it might $A_2\ \mathsf{fconf}^{0.9}_{\bar{\wedge},\triangle}\ train$. Furthermore, if the loop continues with time measurements outside the kernel of the constraints, like in the trace above, the global credibility continuously decreases under the Łukasiewicz t-norm and the product t-norm even if each single credibility is above any threshold. Thus, there will be no $\alpha \in [0,1]$ such that $A_2\ \mathsf{fconf}^{\alpha}_{\curlywedge,\triangle}\ train$ or $A_2\ \mathsf{fconf}^{\alpha}_{\bigstar,\triangle}\ train$. On the contrary, if each single credibility is above the 0.9 threshold, we could have that $A_2\ \mathsf{fconf}^{0.9}_{\bar{\wedge},\triangle}\ train$.

5 Transforming Fuzzy Automata

In this Section we are going to present a transformation from fuzzy-timed automaton into ordinary timed automaton. This transformation is used to characterize the fuzzy relations in terms of ordinary timed automata. Hence, we can take advantage of all the theory and tools developed within the framework of timed automata. The basic idea is to apply a syntactical α-cut to the automaton constraints. Let us first remark that this α-cut does not work in the general case. To understand the reason let us consider the following example.

Example 3. Let us consider the fuzzy relation $C = \overline{x \le 5}^{0.2} \bigstar \overline{y \le 7}^{0.4}$. Then the set $\mathsf{cut}_{0.8}(C)$ cannot be expressed in terms of inequalities. We have

$$\begin{array}{rl}\mathsf{cut}_{0.8}(C) = & \{(a,b) \mid a \le 5 \wedge b \le 7\} \cup \\ & \{(a,b) \mid a \le 5 \wedge 7 < b \le 7.32\} \cup \\ & \{(a,b) \mid 5 < a \le 5.16 \wedge b \le 7\} \cup \\ & \{(a,b) \mid 5.2 \ge a > 5 \wedge 7.4 \ge b > 7 \wedge (1-\frac{a-5}{0.2})(1-\frac{b-7}{0.4}) \ge 0.8\}\end{array}$$

This set cannot be expressed in terms of conjunctions and inequalities so it cannot be part of a constraint of a timed automata.

But If we take the Gödel t-norm instead of the product t-norm we have

$$\mathsf{cut}_{0.8}(\overline{x \le 5}^{0.2}\ \bar{\wedge}\ \overline{y \le 7}^{0.2}) = \{(a,b) \mid a \le 5.12 \wedge b \le 7.32\}$$

That can be expressed in terms of inequalities and conjunctions: $x \le 5.16 \wedge y \le 7.32$. Strictly speaking, this is not a timed constraint expression since 5.32 and 7.36 are not integers. This is not a real problem since the locations and actions are finite sets, therefore the amount of numbers like those is finite. Hence, in order to consider them integers it is enough to consider a time change unit. □

This example shows that if want to make a transformation that keeps the α-cuts, we have to restrict ourselves to fuzzy constraints where the only appearing t-norm is the Gödel one.

Definition 11. *A* fuzzy-Gödel constraint *is a formula generated by the following B.N.F.:*

$$C ::= \mathsf{True} \mid C_1 \,\bar{\wedge}\, C_2 \mid \overline{x \bowtie n}^{\lambda} \mid \overline{x - y \bowtie n}^{\lambda}$$

where $\triangle$ is a t-norm, $\bowtie \in \{\leq, =, \geq\}$, $x, y \in \mathsf{Clocks}$, λ is a non-negative rational number, and $n \in I\!N$.

Let fA be a fuzzy-timed automaton. We say that is a Gödel fuzzy-timed automaton *if all the fuzzy constraints are fuzzy-Gödel constraints.* □

For fuzzy-Gödel constraints we can generalize what we have observed in Example 3. We first define the syntactical α-cut of such constraints

Definition 12. *Let C be a fuzzy-Gödel constraint and let α be a rational number in the interval $[0, 1]$, then we inductively define the* α-cut of C *as:*

$$\mathsf{cut}_\alpha(C) = \begin{cases} \mathsf{True} & \text{if } C = \mathsf{True} \\ x \bowtie n + \alpha \cdot \lambda & \text{if } C = \overline{x <= n}^{\lambda}, \bowtie \in \{\leq, =, \geq\} \\ x - y \bowtie n + \alpha \cdot \lambda & \text{if } C = \overline{x - y <= n}^{\lambda}, \bowtie \in \{\leq, =, \geq\} \\ C_1 \wedge C_2 & \text{if } C = C_1 \,\bar{\wedge}\, C_2 \end{cases}$$

□

Now we can prove that $\mathsf{cut}_\alpha(C)$ is equivalent to the α-cut of the fuzzy contraint C. That is, the set of clocks that give a satisfaction grade greater than α is the same that makes the formula $\mathsf{cut}_\alpha(C)$ true.

Proposition 2. *Let C be a fuzzy-Gödel constraint and let α be a rational number in the interval $[0, 1]$. Then $u \vDash \mathsf{cut}_\alpha(C)$ if and only if $\mu_C(u) \geq \alpha$.*
Proof. It is straightforward by structural induction of C. □

Now that we have shown that the Gödel-fuzzy constraints behave properly when applying the α-cuts, we can extend the concept to Gödel fuzzy automata.

Definition 13. *Let $fA = (L, l_0, E, I)$ be a Gödel fuzzy-timed automaton and let α be a rational number in the interval $[0, 1]$. We define the automaton $\mathsf{cut}_\alpha(fA)$ as the automaton $(L, l_0, E_\alpha, I_\alpha)$, where E_α and I_α are defined as:*

- $(l, a, \mathsf{cut}_\alpha(C), r, l') \in E_\alpha$ *iff* $(l, a, C, r, l') \in E$.
- *If* $I_\alpha(l) = \mathsf{cut}_\alpha(I(l))$. □

First we want to prove that these transformations are correct, that is the obtained automaton conforms the fuzzy specification. The conformance relations in Definition 10 need two t-norms, one to define the operational semantics of fuzzy-timed automata, and another one to combine the values of the operational semantics. Again, the t-norms we need are the Gödel t-norm. To prove it we first need an auxiliary lemma that relates the transitions of a fuzzy-timed automata and the ones of its α-cut.

Proposition 3. *Let $fA = (L, l_0, E, I)$ be a Gödel fuzzy-timed automaton and let α be a rational number in the interval $[0, 1]$. Then:*

1. *$(l, u) \xrightarrow{d} (l, u+d)$ is a transition of the operational semantics of $\mathsf{cut}_\alpha(fA)$ if and only if there exists $\alpha' \geq \alpha$ such that $(l, u) \xrightarrow{d}_{\alpha'} (l, u+d)$ is a transition of the $\overline{\wedge}$-operational semantics of fA.*
2. *$(l, u) \xrightarrow{a} (l', u')$ is a transition of the operational semantics of $\mathsf{cut}_\alpha(fA)$ if and only if there exists $\alpha' \geq \alpha$ such that $(l, u) \xrightarrow{a}_{\alpha'} (l', u')$ is a transition of the $\overline{\wedge}$-operational semantics of fA.*

Proof. The first thing we have to take into account is that the time constraints of $\mathsf{cut}_\alpha(fA)$ are obtained by applying the α-cuts. Hence the invariant of a location l in the automaton $\mathsf{cut}_\alpha(fA)$ is $I_\alpha(l) = \mathsf{cut}_\alpha(I(l))$. Also, by Proposition 2 we have $\mu_{I(l)}(u) \geq \alpha$ if and only if $u \vDash I_\alpha(l)$. So we have part 1.

For part 2, we also have to consider that the transitions of the automaton $\mathsf{cut}_\alpha(fA)$ have the form $(l, a, \mathsf{cut}_\alpha(C), r, l')$ where (l, a, C, r, l') is a transition of fA. Let us recall that we have consider the Gödel t-norm to build the operational semantics of fA therefore $(l, u) \xrightarrow{a}_{\alpha'} (l', u')$ if and only if $\alpha' = \min(\mu_{I(l')}(u[r]), \mu_{I(l)}(u))$. Therefore $\mu_{I(l')}(u[r]) \geq \alpha' \geq \alpha$ and $\mu_{I(l)}(u) \geq \alpha' \geq \alpha$. Again by Proposition 2, that is true if and only if $u[r] \vDash \mathsf{cut}_\alpha(I(l'))$ and $u \vDash \mathsf{cut}_\alpha(I(l))$. □

Proposition 4. *Let fA be a Gödel fuzzy-timed automaton and let α be a rational number in the interval $[0, 1]$. Then $\mathsf{cut}_\alpha(fA)$ $\mathsf{fconfm}^{\alpha}_{\wedge,\overline{\wedge}}$ fA*

Proof. From the previous proposition we have $(d_1, a_1) \cdots (d_n, a_n) \in \mathsf{tr}(\mathsf{cut}_\alpha(fA))$ if and only if there exist $\alpha_i, \beta_i \geq \alpha$ such that $(d_1, \alpha_1, a_1, \beta_1) \cdots (d_n, \alpha_n, a_n, \beta_n) \in \mathsf{ftr}_{\overline{\wedge}}(fA)$. From which the result is immediate. □

From this proposition we get the main result of this Section: in order to prove if some automaton conforms a fuzzy specification it is enough to consider the α-cut of the fuzzy automaton.

Theorem 1. *Let A be an timed automaton, fA be a Gödel fuzzy-timed automaton, and $\alpha \in [0, 1]$. Then*

- *A $\mathsf{fconf}^{\alpha}_{\wedge,\overline{\wedge}}$ fA iff $A \leq_{tr} \mathsf{cut}_\alpha(fA)$*
- *A $\mathsf{fconfm}^{\alpha}_{\wedge,\overline{\wedge}}$ fA iff $A \leq_{tr} \mathsf{cut}_\alpha(fA)$ and $\mathsf{cut}_1(fA) \leq_{tr} A$.* □

6 Conclusions

In this paper, we have introduced a formalism that deals with time by using fuzzy logic. In our opinion, this is an important feature since it is difficult (if not impossible) to have a perfect and exact measurement of time. There have been many issues about the nature of time in formal methods, specially if it should be consider a discrete or dense entity. These issues do not have sense in a fuzzy framework because we are assuming, since the beginning, that we do not have

an exact measurement of time. Although these measurements are imprecise, the fuzzy framework provides a precise measurement of this imprecision.

This work has been focused in the automata framework. As we have mentioned in the introduction, this fuzzy logic has been introduced successfully in the literature. However, as far as we know this is the first time where the fuzziness has been introduce in the time domain of the automata. We have defined the concept fuzzy-timed automata and defined a conformance relation. The concept of t-norm plays a central role in both the definition of the automata and the conformance relation. Among the different t-norms we have discover that the Gödel t-norm is specially important. By using it, we can apply a syntactical α-cut to the fuzzy-timed automata and reduce the fuzzy conformance relations to conformance relations between ordinary timed-automata.

This paper has a number of open loose ends that we plan to address in the future. In first place, the conformance relations are not completely satisfactory. We have introduced two conformance relations: $\mathsf{fconf}^{\alpha}_{\triangle_1,\triangle_2}$ and $\mathsf{fconfm}^{\alpha}_{\triangle_1,\triangle_2}$. The first one corresponds directly to trace inclusion while the second one is closer to trace equivalence. The first one has the same problems as the trace inclusion; while the second one has a conceptual problem because we require that certain fuzzy formula has a satisfaction grade of 1. We do not think this is very adequate in a fuzzy environment and we are working in an alternate possibility.

Another important issue to address is to implement a software tool. In Section 5, we have reduced the fuzzy relations to relation between ordinary automata. So, we can take advantage of the already existing software tools to verify the fuzzy relations, we are already working in a tool that performs this task. The problem is that to perform the transformation we can only work with Gödel t-norms. Apart from the fact that we would like to work with another t-norms, we think we can address this problem in an alternative way. The existing tools for timed automata like Uppaal [4] perform a discretization of time to transform a timed automaton into a non-timed automaton. What we plan is to take advantage of the fuzziness to try to avoid the discretization of time.

Acknowledgments. We like to express our gratefulness to the anonymous reviewers for their valuable comments. We also want to apologize to them for the bad English of the first submission. We hope we have improved it.

References

1. Agrawal, M., Thiagarajan, P.S.: The discrete time behavior of lazy linear hybrid automata. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 55–69. Springer, Heidelberg (2005)
2. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking for probabilistic real-time systems (extended abstract). In: Leach-Albert, J., et al. (eds.) [10], pp. 115–126
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

5. Davies, J., Schneider, S.: A brief history of timed CSP. Theor. Comput. Sci. 138(2), 243–271 (1995)
6. Doostfatemeh, M., Kremer, S.C.: General fuzzy automata, new efficient acceptors for fuzzy languages. In: IEEE International Conference on Fuzzy Systems, pp. 2097–2103. IEEE, Los Alamitos (2006)
7. Fernández-Vilas, A., Pazos-Arias, J.J., Díaz-Redondo, R.P.: Extending timed automaton and real-time logic to many-valued reasoning. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 185–204. Springer, Heidelberg (2002)
8. Hennessy, M., Regan, T.: A process algebra for timed systems. Information and Computation 117(2), 221–239 (1995)
9. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Verifying quantitative properties of continuous probabilistic timed automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 123–137. Springer, Heidelberg (2000)
10. Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.): ICALP 1991. LNCS, vol. 510. Springer, Heidelberg (1991)
11. Mensch, S.I., Lipp, H.M.: Fuzzy specification of finite state machines. In: Adshead, G., Jess, J.A.G. (eds.) EURO-DAC, pp. 622–626. IEEE Computer Society, Los Alamitos (1990)
12. Merayo, M.G., Núñez, M., Rodríguez, I.: Implementation relations for stochastic finite state machines. In: Horváth, A., Telek, M. (eds.) EPEW 2006. LNCS, vol. 4054, pp. 123–137. Springer, Heidelberg (2006)
13. Mraz, M., Lapanja, I., Zimic, N., Virant, J.: Fuzzy numnbers as input to fuzzy automata. In: IEEE, pp. 453–456. IEEE, Los Alamitos (1999)
14. Nguyen, H.T., Walker, E.A.: A first course in fuzzy logic. CRC Press, Inc., Boca Raton (1996)
15. Nicollin, X., Sifakis, J.: The algebra of timed processes, ATP: Theory and application. Information and Computation 114(1), 131–178 (1994)
16. Ortega-Mallén, Y.: Operational semantics for timed observations. In: Vytopil, J. (ed.) FTRTFT 1992. LNCS, vol. 571, pp. 507–527. Springer, Heidelberg (1991)
17. Springintveld, J., Vaandrager, F.W., D'Argenio, P.R.: Testing timed automata. Theoretical Computer Science 254(1-2), 225–257 (2001)
18. Wee, W.G., Fu, K.S.: A formulation of fuzzy automata and its application as a model of learning systems. IEEE Trans. on Systems, Man and Cybernetics 5(3), 215–223 (1969)
19. Yi, W.: Ccs + Time = an interleaving model for real time systems. In: Leach-Albert, J., et al. (eds.) [10], pp. 217–228
20. Zadeh, L.A.: Fuzzy sets. Information and Control 8(3), 338–353 (1965)
21. Zadeh, L.A.: Fuzzy logic and its application to approximate reasoning. In: IFIP Congress, pp. 591–594 (1974)

Model Checking of Hybrid Systems Using Shallow Synchronization*

Lei Bu[1], Alessandro Cimatti[2], Xuandong Li[1], Sergio Mover[2], and Stefano Tonetta[2]

[1] State Key Laboratory for Novel Software Technology, Nanjing University
[2] Fondazione Bruno Kessler - IRST

Abstract. Hybrid automata are a widely accepted modeling framework for systems with discrete and continuous variables. The traditional semantics of a network of automata is based on interleaving, and requires the construction of a monolithic hybrid automaton based on the composition of the automata. This destroys the structure of the network and results in a loss of efficiency, especially using bounded model checking techniques. An alternative compositional semantics, called "shallow synchronization", exploits the locality of transitions and relaxes time synchronization. The semantics is obtained by composing traces of the local automata, and superimposing compatibility constraints resulting from synchronization.

In this paper, we investigate the different symbolic encodings of the reachability problem of a network of hybrid automata. We propose a novel encoding based on the shallow synchronization semantics, which allows different strategies for searching local paths that can be synchronized. We implemented a bounded reachability search based on the use of an incremental Satisfiability-Modulo-Theory solver. The experimental results confirm that the new encoding often performs better than the one based on interleaving.

1 Introduction

Hybrid automata ([13]) are increasingly recognized as a clean modeling framework for systems with discrete and continuous variables. Many systems are structured into components, and can often be naturally modeled as networks of communicating hybrid automata: local activities of each component amount to transitions local to each hybrid automaton; communications and other events that are shared between/visible for various components are modeled as synchronizing transitions of the automata in the network; time elapse is modeled as shared timed transitions. The traditional asynchronous semantics is based on interleaving, and requires the construction of a monolithic hybrid automaton based on the composition of the automata in the network. Intuitively, this means that a path in the automaton is the result of the composition of interleaving paths. However, the monolithic automaton resulting from the composition can be seen

* The authors at Nanjing University are supported by the National Natural Science Foundation of China (No.90818022 and No.60721002) and the National 863 High-Tech Programme of China (No.2009AA01Z148). A. Cimatti is supported by the European Commission (FP7-2007-IST-1-217069 COCONUT and ACP7-GA-2008-212088 MISSA) S. Tonetta is supported by the Provincia Autonoma di Trento (project ANACONDA).

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 155–169, 2010.
© IFIP International Federation for Information Processing 2010

as the result of a "strict synchronization", and the analysis has to deal with an overly large number of paths, since the structure and the locality of the network are not taken into account.

An alternative semantics [5] for networks of automata exploits the fact that automata can be "shallowly synchronized". The intuition is that each automaton can proceed based on its individual "local time scale", unless they perform a synchronizing transition, in which case they must realign their absolute time. This results in a more concise semantics, where traces of the network are obtained by composing traces of local automata, each with local time elapse, by superimposing structure based on shared communication.

In this paper, we provide a fully symbolic account for bounded reachability under "shallow synchronization", and we explore various search strategies. We implement the approach in the sub-case of linear hybrid automata and we use a Satisfiability-Modulo-Theory (SMT) [15] solver to check the satisfiability of the formulas encoding the reachability problem. The main advantage is that the transition relation of each automata is unrolled only for the steps necessary to reach locally the target (regardless the length of the interleaving with the other automata). Typically, local paths are much shorter because they do not need to stutter allowing other processes to perform local or non-shared events. The disadvantage is that we may use additional variables and constraints. We experimentally investigate this trade-off and the results show that the new encoding often performs better than the one based on interleaving. In particular, the improvement increases at the growth of the difference between the length of the local traces and the length of the interleaving trace.

The paper is structured as follows. In Section 2 we present some background on hybrid automata and their composition through interleaving. In Section 3 we present the shallow synchronization semantics revising the concepts described in [5] and defining explicit mappings from the semantics with strict synchronization to the shallowly synchronized one, and vice versa. In Section 4 we show several ways to symbolically encode the bounded model checking problem for shallow synchronization semantics. In Section 5 we discuss related work. In Section 6 we experimentally evaluate our approach. In Section 7 we draw some conclusions.

2 Background

Notation. Given a set V of real-valued variables, we denote with $\mathcal{LB}(V)$ the set of Boolean combinations of linear equalities and inequalities over V. We denote with V' the set of "next" variables and with $\dot{V}$ the set of first derivative of the variables in V over time. We write $V' = V$ as an abbreviation for $\bigwedge_{v \in V} v' = v$.

If f is a collection of real functions $\{f^v\}_{v\in V}$, we denote with f^V the composed function $f^V(t) = \Pi_{v\in V} f^v(t)$.

Given a formula ϕ in $\mathcal{LB}(V)$ and $\overline{V}$ a set of copies of the variables in V, we denote with $\phi(\overline{V})$ the formula obtained by substituting each $v \in V$ with its copy $\overline{v} \in \overline{V}$. Given a formula ϕ in $\mathcal{LB}(\dot{V})$, two copies $\overline{V}_1$ and $\overline{V}_2$ of V, and ψ a linear term, we denote with $\phi(\frac{\overline{V}_2 - \overline{V}_1}{\psi})$ the formula obtained by substituting each $\dot{v} \in \dot{V}$ with $\frac{\overline{v}_2 - \overline{v}_1}{\psi}$ and then multiplying by ψ (thus $\phi(\frac{\overline{V}_2 - \overline{V}_1}{\psi})$ is a Boolean combination of linear constraints).

2.1 Hybrid Automata

Due to lack of space, but without loss of generality, we restrict the presentation to the framework of Linear Hybrid Automata (LHA). The results presented in the rest of this paper however apply to the general case of Hybrid Automata as defined in [13][1].

Definition 1 ([13]). *A* LHA *is a tuple* $\langle Q, E, X, F, I, Z, J, U, L\rangle$ *where*

- *Q is the set of locations,*
- *$E \subseteq Q \times Q$ is the set of edges,*
- *X is the set of continuous variables,*
- *for each $q \in Q$, $F(q) \in \mathcal{LB}(\dot{X})$ is the flow condition (denoted also as F_q),*
- *for each $q \in Q$, $I(q) \in \mathcal{LB}(X)$ is the initial condition (denoted also as I_q),*
- *for each $q \in Q$, $Z(q) \in \mathcal{LB}(X)$ is the invariant condition (denoted also as Z_q),*
- *for each $e \in E$, $J(e) \in \mathcal{LB}(X \cup X')$ is the jump condition (denoted also as J_e),*
- *U is the set of labels,*
- *for each $e \in E$, $L(e) \in U$ is the label of the edge (denoted also as L_e).*

Definition 2. *A* run *of a LHA H is a sequence $\langle q_0, s_0\rangle \xrightarrow{a_1} \langle q_1, s_1\rangle \ldots \langle q_{n-1}, s_{n-1}\rangle \xrightarrow{a_n} \langle q_n, s_n\rangle$ such that:*

- *for all i, $0 \le i \le n$, $q_i \in Q$ and s_i is an assignment to the variables of X;*
- *for all i, $1 \le i \le n$, $a_i \in U \cup \mathbb{R}^{\ge 0}$; hereafter $t_i = \sum_{1\le j\le i, a_j \in \mathbb{R}^{\ge 0}} a_j$ and $t_0 = 0$; we call t_n the final time of the run; we call the pair $\langle a_i, t_i\rangle$ an* event*;*
- *for all i, $1 \le i \le n$, if $a_i \in \mathbb{R}^{\ge 0}$, then $q_{i-1} = q_i$ and there exists a collection of real functions $\{f_i^x\}_{x\in X}$ such that f_i^x is differentiable over $[t_{i-1}, t_i]$ and $f_i^X(t_{i-1}) = s_{i-1}$ and $f_i^X(t_i) = s_i$;*
- *for all i, $1 \le i \le n$, if $a_i \in U$ then $\langle q_{i-1}, q_i\rangle \in E$ and $a_i = L(\langle q_{i-1}, q_i\rangle)$;*
- *for all i, $1 \le i \le n$, if $a_i \in \mathbb{R}^{\ge 0}$, then for all $t \in [t_{i-1}, t_i]$, then $\dot{f}_i^X(t) \models F_{q_i}$;*
- *$s_0 \models I_{q_0}$ and for all i, $0 \le i \le n$, $s_i \models Z_{q_i}$;*
- *for all i, $1 \le i \le n$, if $a_i \in \mathbb{R}^{\ge 0}$, then for all $t \in [t_{i-1}, t_i]$, $f_i^X(t) \models Z_{q_i}$;*
- *for all i, $1 \le i \le n$, if $a_i \in U$ then $s_{i-1}, s_i \models J_{\langle q_{i-1}, q_i\rangle}$.*

A run σ_1 is a refinement of another run σ_2 iff σ_1 is obtained by σ_2 by splitting some timed transition $\langle q_i, s_{i-1}\rangle \xrightarrow{a_i} \langle q_i, s_i\rangle$, $a_i \in \mathbb{R}^{\ge 0}$ into two or more timed transitions $\langle q_i, s_{i-1}\rangle \xrightarrow{a_{i_1}} \langle q_i, s_{i_1}\rangle \ldots \langle q_i, s_{i_{h-1}}\rangle \xrightarrow{a_{i_h}} \langle q_i, s_i\rangle$ such that $a_{i_j} \in \mathbb{R}^{\ge 0}$, $1 \le j \le h$, and, $\sum_{1\le j\le h} a_{i_j} = a_i$. A timed transition $\langle q_i, s_{i-1}\rangle \xrightarrow{a_i} \langle q_i, s_i\rangle$ with $a_i = 0 \in \mathbb{R}^{\ge 0}$ is called a stuttering transition.

2.2 Network of Hybrid Automata

The definition of network of hybrid automata is based on the definition in [13], which means components communicate with each other by shared labels.

[1] As far as the solutions of the flow conditions can be represented in the logic handled by the SMT solver.

Definition 3. *Given two LHAs* $H_1 = \langle Q_1, E_1, X_1, F_1, I_1, Z_1, J_1, U_1, L_1\rangle$ *and* $H_2 = \langle Q_2, E_2, X_2, F_2, I_2, Z_2, J_2, U_2, L_2\rangle$ *with* $Q_1 \cap Q_2 = X_1 \cap X_2 = \emptyset$*, the* composition $H_1 \times H_2$ *is the LHA* $\langle Q_P, E_P, X_P, F_P, I_P, Z_P, J_P, U_P, L_P\rangle$ *where*

- $Q_P = Q_1 \times Q_2$,
- $E_P = \{\langle q_1 \times q_2, q'_1 \times q'_2\rangle \mid$ *either* $\langle q_1, q'_1\rangle \in E_1, q_2 = q'_2, L_1(\langle q_1, q'_1\rangle) \notin U_2$, *or* $\langle q_2, q'_2\rangle \in E_2, q_1 = q'_1, L_2(\langle q_2, q'_2\rangle) \notin U_1$, *or* $\langle q_1, q'_1\rangle \in E_1, \langle q_2, q'_2\rangle \in E_2, L_1(\langle q_1, q'_1\rangle) = L_2(\langle q_2, q'_2\rangle)\}$,
- $X_P = X_1 \cup X_2$,
- $F_P(q_1 \times q_2) = F_1(q_1) \wedge F_2(q_2)$,
- $I_P(q_1 \times q_2) = I_1(q_1) \wedge I_2(q_2)$,
- $Z_P(q_1 \times q_2) = Z_1(q_1) \wedge Z_2(q_2)$,
- $U_P = U_1 \cup U_2$,
- $J_P(\langle q_1 \times q_2, q'_1 \times q'_2\rangle) = \begin{cases} J(\langle q_1, q'_1\rangle) \wedge X'_2 = X_2 & \text{if } q_2 = q'_2, L_1(\langle q_1, q'_1\rangle) \notin U_2 \\ J(\langle q_2, q'_2\rangle) \wedge X'_1 = X_1 & \text{if } q_1 = q'_1, L_2(\langle q_2, q'_2\rangle) \notin U_1 \\ J(\langle q_1, q'_1\rangle) \wedge J(\langle q_2, q'_2\rangle) & \text{if } L_1(\langle q_1, q'_1\rangle) = L_2(\langle q_2, q'_2\rangle), \end{cases}$
- $L_P(\langle q_1 \times q_2, q'_1 \times q'_2\rangle) = \begin{cases} L(\langle q_1, q'_1\rangle) \text{ if } q_2 = q'_2, L_1(\langle q_1, q'_1\rangle) \notin U_2 \\ L(\langle q_2, q'_2\rangle) \text{ if } q_1 = q'_1, L_2(\langle q_2, q'_2\rangle) \notin U_1 \\ L(\langle q_1, q'_1\rangle) \text{ if } L_1(\langle q_1, q'_1\rangle) = L_2(\langle q_2, q'_2\rangle). \end{cases}$

Definition 4. *A* network $\mathcal{H}$ *of LHAs is a tuple of LHAs.*

The semantics of a network of automata is given by the composition of the automata.

Definition 5. *A* synchronized run *of a network* $\mathcal{H} = \langle H_1, \ldots, H_n\rangle$ *is a run of the composition* $H_1 \times \ldots \times H_n$*.*

In the following we refer to a run of a single automaton in a network as "local", to distinguish it from a run of the composition automaton.

Reachability problem. Given a network of automata $\mathcal{H} = \langle H_1, H_2, \ldots, H_n\rangle$, and a target set $T = \langle q_1, q_2, \ldots, q_n\rangle$, the reachability problem for $\mathcal{H}$ and T is to verify whether $q_1 \times q_2 \times \cdots \times q_n$ can be reached in the composition $\mathcal{H}$. Thus, we consider only finite runs, although the approach can be extended to infinite runs which can be represented by lasso-shape paths.

3 Shallow Synchronization Semantics

While in strict synchronization the behavior of a network is basically obtained by interleaving, in shallow synchronization a run of the network is the result of "composition" of runs local to each automaton in the network. The intuition is demonstrated in Figure 1. In the upper part, we see three traces of three automata in a network. Each automaton H_i has a local label τ; the ij labels are shared between processes H_i and H_j; δ denotes local time elapse. We notice that the synchronization over the ij labels happens exactly at the same time, e.g., 12 takes place at absolute time 5, although the number of transitions required by H_1 and H_2 is different. In the lower part of the figure, we report the corresponding trace based on interleaving (where each box contains the

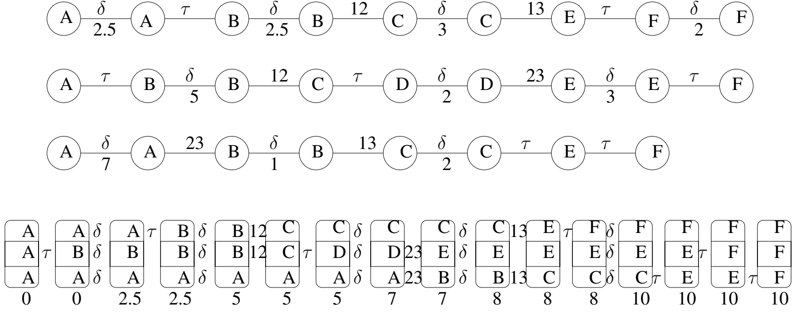


Fig. 1. Three local traces (above), and the corresponding interleaving (below)

state of each of the three processes). Stuttering (e.g. of process 1 and 3 in the first step) is modeled by the fact that a process does not have any label on its side.

We also define a mapping of a set of shallowly synchronized runs of the automata into a run in the composition of the automata. Intuitively, the mapping induces an equivalence relation among the runs of the composition automaton which are obtained by composing the same set of local runs with different interleaving. The shallow synchronization is defined according to the trace of a run i.e., the list of events occurring in the run. An S-trace, with $S \subseteq U$, is a trace restricted to the labels in S.

Definition 6. *Given a set of labels $S \subseteq U$ and a run $\sigma = \langle q_0, s_0 \rangle \xrightarrow{a_1} \langle q_1, s_1 \rangle \dots \xrightarrow{a_n} \langle q_n, s_n \rangle$, the S-*trace *$\tau_S(\sigma)$ is the sequence of events $\langle a_1, t_1 \rangle, \langle a_2, t_2 \rangle, \dots, \langle a_k, t_k \rangle$ where t_i is the time at which the event a_i occurs in σ.*

Definition 7. *Given two LHAs H_1 and H_2 with sets of labels U_1 and U_2 resp., let σ_1 be a run of H_1 and σ_2 a run of H_2. Let S be the intersection of U_1 and U_2 ($S = U_1 \cap U_2$). The pair $\langle \sigma_1, \sigma_2 \rangle$ is* consistent *iff the S-trace of σ_1 is equal to the S-trace of σ_2 ($\tau_S(\sigma_1) = \tau_S(\sigma_2)$) and the final time of σ_1 is equal to the final time of σ_2.*

The last constraint on the final time is necessary because otherwise the two runs may terminate with a series of local steps with different timings.

Definition 8. *A* shallowly synchronized run *of a network of LHAs is a tuple $\theta = \langle \sigma_1, \dots, \sigma_n \rangle$ such that σ_j is the run of H_j and, for all j, h, $1 \leq j < h \leq n$, σ_j and σ_h are consistent.*

If θ is a shallowly synchronized run, we denote with θ_j the j-th component of θ.

Remark 1. In general, two different events can occur at the same time in the same run, because discrete transitions are not forced to be interleaved with timed transitions. Moreover, simultaneous events may be interleaved with different orders.

However, in many cases, we can assume that whenever two events occur simultaneously, they have a fixed order. Then, the pair $\langle \sigma_1, \sigma_2 \rangle$ is consistent simply iff for all $a \in U_1 \cap U_2$ and $t \in \mathbb{R}$, $\langle a, t \rangle$ occurs in σ_1 iff $\langle a, t \rangle$ occurs in σ_2. I.e., having the events

at the same time guarantees that the traces are the same. The definitions and theorems in [5] have this assumption, while in this section we consider the most general case.

Projection of a synchronized run of the composition automaton on one component is the corresponding run local to that component automaton. Intuitively, the set of projections of a synchronized run form a shallowly synchronized run. The projection induces an equivalence relation over strictly synchronized traces, namely the equivalence of runs that are the same modulo a reordering of the interleaved labels.

Definition 9. *Given a network $\mathcal{H}$ and an LHA $H \in \mathcal{H}$, the* projection π_H *of a synchronized run σ of $\mathcal{H}$ over H is obtained by projecting the states and the assignments occurring in σ on the H component and substituting transitions labeled with events not accepted by H with stuttering transitions*[2].

The following theorem states the relationship between the two semantics[3].

Theorem 1. *Given a synchronized run σ, the tuple of projections $\langle \pi_{H_1}(\sigma), \ldots, \pi_{H_n}(\sigma) \rangle$ on the different components is a shallowly synchronized run. Vice versa, given a shallowly synchronized run θ, there exists a synchronized run σ such that $\big\langle \pi_{H_1}(\sigma), \ldots, \pi_{H_n}(\sigma) \big\rangle$ is a refinement of θ.*

As corollary, there exists a strictly synchronized run reaching $q_{H_1} \times \cdots \times q_{H_n}$ iff there exists a shallowly synchronized run θ such that for all i, $1 \leq i \leq n$, θ_{H_i} reaches q_{H_i}.

4 Symbolic Encoding

In this section, first, we recall how linear hybrid automata and their reachability problem can be encoded symbolically; second, we show how we can encode symbolically the problem for a network with strict and shallow synchronization.

4.1 Symbolic Encoding for Single Automaton

In the following, in order to encode the flow condition into a quantifier-free formula, we assume the convexity of the invariant conditions. The symbolic encoding of a single LHA consists of three formulas representing respectively the initial, the transition, and the invariant condition. The encoding uses the following additional variables: a discrete variable loc that represents the current location; a real-valued variable δ that represents the time elapsed at the current step; a discrete variable l that represents the label taken at the current step; and two distinguished values T and S, representing a timed transition and stuttering, respectively.

[2] The projection is well defined because if $\langle q_{i-1}, s_{i-1} \rangle \xrightarrow{a_i} \langle q_i, s_i \rangle$ occurs in σ and a_i is not a label of H, then the H components of q_{i-1} and s_{i-1} are equal to the H components of q_i and s_i respectively. Thus, the transition can be locally substituted with a stuttering transition.

[3] An extended version with proofs can be find at http://es.fbk.eu/people/tonetta/papers/forte10/

The encoding consists of the following formulas:

$$\textsc{Init} := \bigwedge_{q\in Q} (loc = q \rightarrow I_q(X))$$

$$\textsc{Invar} := \bigwedge_{q\in Q} (loc = q \rightarrow Z_q(X))$$

$$\textsc{Trans} := \bigwedge_{q\in Q} (loc = q \rightarrow (\textsc{Stutter} \vee \textsc{Timed}_q \vee \bigvee_{(q,p)\in E} \textsc{Untimed}_{q,p}))$$

$$\textsc{Stutter} := l = \textsc{s} \wedge \delta = 0 \wedge loc' = loc \wedge X' = X$$

$$\textsc{Timed}_q := l = \textsc{t} \wedge \delta > 0 \wedge loc' = loc \wedge F_q(\frac{X' - X}{\delta})$$

$$\textsc{Untimed}_{q,p} := l = L_{q,p} \wedge \delta = 0 \wedge loc' = p \wedge J_{q,p}(X, X')$$

Given a reachability problem and a bound k on the length of the runs, we can encode the bounded reachability problem into a formula which is satisfiable iff there exists a run reaching the target condition. We assume to have a formula TARGET encoding the target condition. For example, if we want to check the reachability of the location q, we can set $\textsc{Target} := loc = q$.

As usual in BMC, we introduce $k + 1$ copies of every variable in the encoding of the automata. Then, the reachability problem can be encoded into the following formula:

$$\textsc{Bmc}^k := \textsc{Init}^0 \wedge \textsc{Invar}^0 \wedge \bigwedge_{0\leq i<k} \left(\textsc{Trans}^i \wedge \textsc{Invar}^{i+1}\right) \wedge \textsc{Target}^k$$

where ϕ^i means that the current and next variables of ϕ have been substituted with their i-th and $(i + 1)$-th copy, respectively.

When we consider a network, we use $\textsc{Bmc}^k_H$ to refer to the encoding of the problem for the automaton H.

4.2 Symbolic Encoding Based on Interleaving

In principle, it would be possible to generate the automaton corresponding to the composition of two or more LHAs, and use the above encoding. A more reasonable encoding for a network is based on the encoding of each LHA in the network. The idea is to simply conjunct the encodings forcing the shared event variables to be true exactly at the same steps, and forcing the processes to "stutter" when they are not activated. We assume that the variable δ is shared among the encodings of the different automata.

The reachability problem with a bound k can be encoded as

$$\textsc{BmcInt}^k_{\mathcal{H}} := \bigwedge\nolimits_{1\leq j\leq n} \textsc{Bmc}^k_{H_j} \wedge \textsc{StrictSync}^k_{\mathcal{H}}$$

where STRICTSYNC guarantees that for every pair of processes j and h, every shared event and the timed event occur at the same step in the two processes, and while a non-shared event occurs in one process, the other process must stutter[4]:

[4] Note that it is not necessary to force at least one process not to stutter.

$$\textsc{StrictSync}^k_{\mathcal{H}} := \bigwedge_{1\le j<h\le n} \bigwedge_{0\le i<k} \bigwedge_{a\in U_j\cap U_h} (l^i_j = a \leftrightarrow l^i_h = a)$$
$$\wedge \bigwedge_{a\in U_j\setminus U_h} (l^i_j = a \rightarrow l^i_h = \text{S})$$
$$\wedge \bigwedge_{a\in U_h\setminus U_j} (l^i_h = a \rightarrow l^i_j = \text{S})$$
$$\wedge\ (l^i_h = \text{T} \leftrightarrow l^i_j = \text{T})$$

The encoding is compositional in the sense that each automaton is individually encoded. However, the necessity of stuttering on non-shared events and of performing shared events in the same steps may cause complex runs (as shown in Fig. 1).

We also consider a variant of the above encoding where we allow discrete transitions in different automata to occur at the same step of the encoding. Basically, with this variant, we do not force a process to stutter when other processes perform either a local event or an event which is not shared by the process. In this cases, we omit the constraints which force to stutter. This encoding corresponds to the *step semantics* used in [12] for encoding the bounded model checking problem of asynchronous systems.

4.3 Symbolic Encoding Based on Shallow Synchronization

In this section, we propose an encoding based on shallow synchronization. We let each automaton keep its own copy of the bound k and the elapsed time δ; we do not force processes to stutter and we let shared events occur at different (local) steps. This means that each of the local encodings is able to construct a local trace.

The reachability problem with bounds $\overline{k} = \langle k_1, k_2, \ldots, k_n\rangle$ can be encoded as

$$\textsc{BmcSS}^{\overline{k}}_{\mathcal{H}} := \bigwedge_{1\le j\le n} \textsc{Bmc}^{k_j}_{H_j} \wedge \textsc{ShallowSync}$$

where ShallowSync encodes the constraints enforcing that all the paths must be consistent according to Definition 7. In the following, we present different ways to encode ShallowSync. (We assume to be in the case described in Remark 1, but all the encodings that we are showing can be lifted to the general case.)

Encoding based on enumeration. The first way to encode ShallowSync is by enumerating all possible combinations of steps on which the synchronization occurs. For example, processes P1 and P2 may synchronize over event a, but a may occur in step 2 for P1, and in step 4 for P2. ShallowSync guarantees that, for all pairs of processes, (i) if a shared event occurs in the first process, then the event must occur also in the second process at the same time (possibly in different steps), and (ii) the final time of the two processes is the same:

$$\text{SHALLOWSYNC} := \bigwedge_{1\leq j<h\leq n}\bigwedge_{a\in U_j\cap U_h} \bigwedge_{1\leq i_j\leq k_j} (l_j^{i_j} = a \leftrightarrow \bigvee_{1\leq i_h\leq k_h} l_h^{i_h} = a \wedge t_j^{i_j} = t_h^{i_h}) \wedge$$
$$\bigwedge_{1\leq i_h\leq k_h} (l_h^{i_h} = a \leftrightarrow \bigvee_{1\leq i_j\leq k_j} l_j^{i_j} = a \wedge t_j^{i_j} = t_h^{i_h}) \wedge$$
$$\bigwedge_{1<j\leq n} t_j^{k_j} = t_1^{k_1}$$

Local reasoning. We propose a variant of the previous encoding which can be split into constraints local to each automaton, and one for each step. The encoding uses the following additional variables:

- for each automaton H_j, for each shared label l, a variable $count_{l,j}^i$ to represent how many times l has occurred in H_j before step i;
- for each shared label l, a group of variables $occ_time_{i,l}$ to represent the time at which the i-th occurrence of l is fired;
- for each shared label l, a variable l_{last} to record how many times l has been fired in the whole run;
- c_{last} to record the time at which the system reaches the target.

Note that the variables without superscript are untimed, in the sense that they do not depend on any temporal step.

The shallow synchronization can be encoded as:

$$\text{SHALLOWSYNC} := \bigwedge_{1\leq j\leq n} \bigwedge_{0\leq i<k_j} \text{SHALLOWSTEP}_j^i \wedge$$
$$\text{COUNTERINIT}_j \wedge \left(\bigwedge_{0\leq i<k_j} \text{COUNTERSTEP}_j^i \right) \wedge \text{FINALSHALLOW}_j$$

where SHALLOWSTEP_j^i states that if in the i-th step, an event l occurs in the j-th process for the g-th time, then the local time of the process must be $occ_time_{g,l}$:

$$\text{SHALLOWSTEP}_j^i := \bigwedge_{l\in U_j} (l_j^i = l) \rightarrow \bigwedge_{1\leq g\leq i} ((count_{l,j}^i = g) \rightarrow t_j^i = occ_time_{g,l})$$

COUNTERINIT and COUNTERSTEP encode how the counters evolve:

$$\text{COUNTERSTEP}_j^i := \bigwedge_{l\in U_j} (l_j^i = l) \rightarrow (count_{l,j}^{i+1} = count_{l,j}^i + 1)$$
$$\text{COUNTERINIT}_j := (count_{l,j}^0 = 0)$$

while FINALSHALLOW states that the final values of the counters and the local time must be the same:

$$\text{FINALSHALLOW}_j := (\bigwedge_{l\in U_j} count_{l,j}^{k_j} = l_{last}) \wedge (t_j^{k_j+1} = c_{last}))$$

Exploiting richer theories. It is possible to represent the above encoding with richer theories introducing uninterpreted functions symbols. In particular we represent the

time of the i-th occurrence of a label l as a function occ_time_l from integers to reals. This way we can rewrite SHALLOWSTEP into

$$\text{SHALLOWSTEP}_j^i := \bigwedge_{l \in U_j} (l_j^i = l) \rightarrow (t_j^i = occ_time_l(count_{l,j}^i)).$$

5 Related Work

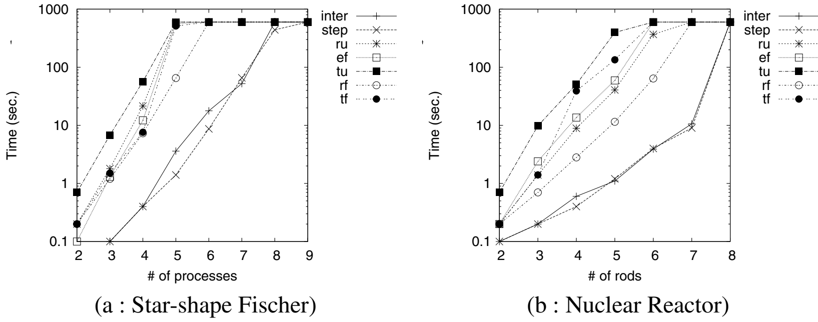
The shallow semantics (defined in [5] and adopted in this paper) bears many similarities with the "local-time" semantics defined in [3] for networks of timed systems and can in fact be seen as a generalization to the hybrid case of [3]. Indeed, neither requires the synchronization of timed transitions of different components; they both use local clocks that are re-synchronized upon shared events. The two semantics differ in the types of runs used to solve the reachability problem: the shallow semantics consists of sets of local runs, while the local-time semantics consists of runs in the interleaving composition. With a mapping similar to the one defined in Section 3, it can be shown that the two semantics are equivalent. As far as we know, this is the first attempt to exploit the shallow/local-time semantics to improve BMC.

Partial-Order Reduction (POR) [11] is one of the most known and used technique to tackle the state-space explosion problem due to interleaving of concurrent systems. The idea is to identify cases when the order of transitions is not relevant in order to prune the search space. The application of POR techniques is difficult in the context of timed and hybrid systems because the timed transitions are global actions which typically interleave the local transition, and thus forbid the pruning performed by POR. The local-time semantics was proposed in [3] to enable POR by removing the synchronization on timed transitions. Other works as in [17] propose symbolic versions of POR and combine them with bounded model checking and SMT. The main difference between POR and the techniques presented in this paper is that while POR tackles the interleaving explosion problem by fixing the order of independent transitions, we allow them to be executed in parallel.

Also related is the "step" semantics, used in [12] for an efficient encoding of the reachability problem in a network of asynchronous systems. The work in [12] is limited to the case of discrete transitions. The idea presented in this paper can be seen as a generalization of the step semantics to the case of timed transitions.

The work described in [16] proposes an event-order abstraction to verify timed automata. The idea is to analyze the discrete and continuous aspects separately by first finding a discrete path causing an error and then computing a set of timing constraints that make the path realistic. Similarly, CEGAR-based approaches such as [1,14] perform a search on a purely discrete abstraction of the hybrid automaton, and check if the obtained paths are compliant with the original constraints.

The first approach that adopts a shallowly synchronized semantics is presented in [5] for path-oriented bounded reachability analysis of a network of LHAs. In the approach, one path is selected for each component and all selected paths compose a path set for reachability analysis. Each path is independently encoded to a set of constraints while synchronization controls are encoded according to the position of shared labels. By merging all the constraints, the path-oriented reachability problem can be transformed



(a : Star-shape Fischer) (b : Nuclear Reactor)

Fig. 2. Results where the length of a local run depends on the number of processes

to the feasibility problem of the resulting linear constraint set, which can be solved by linear programming efficiently. This approach has been extended in BACH [6] into a general bounded reachability analysis technique. Different from the approach presented in this paper, this technique traverses the structure of a network of automata using depth-first search and checks the abstract path set one by one.

In the approaches mentioned above, the search is carried out in two stages: in the first, a discrete abstraction of the problem is constructed, while in the second the candidate paths found in the abstract state are checked for consistency in the concrete space. In our approach, the SMT solver carries out the refinement automatically during the search, on demand. With respect to explicit-state search, the symbolic representation is less sensitive to the state-space explosion problem. With respect to abstraction-based techniques, the BMC technique is more tailored to find error paths.

Bounded model checking for hybrid systems using SMT solvers has been investigated in [2,10,8,9]. The characterizing feature of our work is the attempt to leverage the structure induced by the synchronization of a network of hybrid automata.

6 Experimental Evaluation

6.1 Implementation

We implemented the encodings presented in Section 4 within the setting of NUSMT, a model checker that extends NuSMV2 [7] with SMT techniques. The solver used to check the satisfiability of the formulas was MathSat [4], which provides an incremental interface. Thus, the search interacts with the solver to analyze problems of increasing depth. As standard in bounded model checking, we exploit the fact that subproblems at increasing depth share large parts of the encoding: the solver is able to retain information discovered during the previous searches to solve next subproblems more efficiently. We use the following notation to refer to the options: we use `e` for using the enumerative encoding, `r` for using local reasoning, `t` for using local reasoning with uninterpreted functions; with regards to the incrementality, when we use local reasoning, we can add the synchronization constraints during the unrolling (denoted with `u`) or add them after the unrolling (denoted with `f`). Overall, the options are `ru`, `rf`, `ef`, `tu`, `tf` (e.g., `ru` means using local encoding with the constraints added during the unrolling).

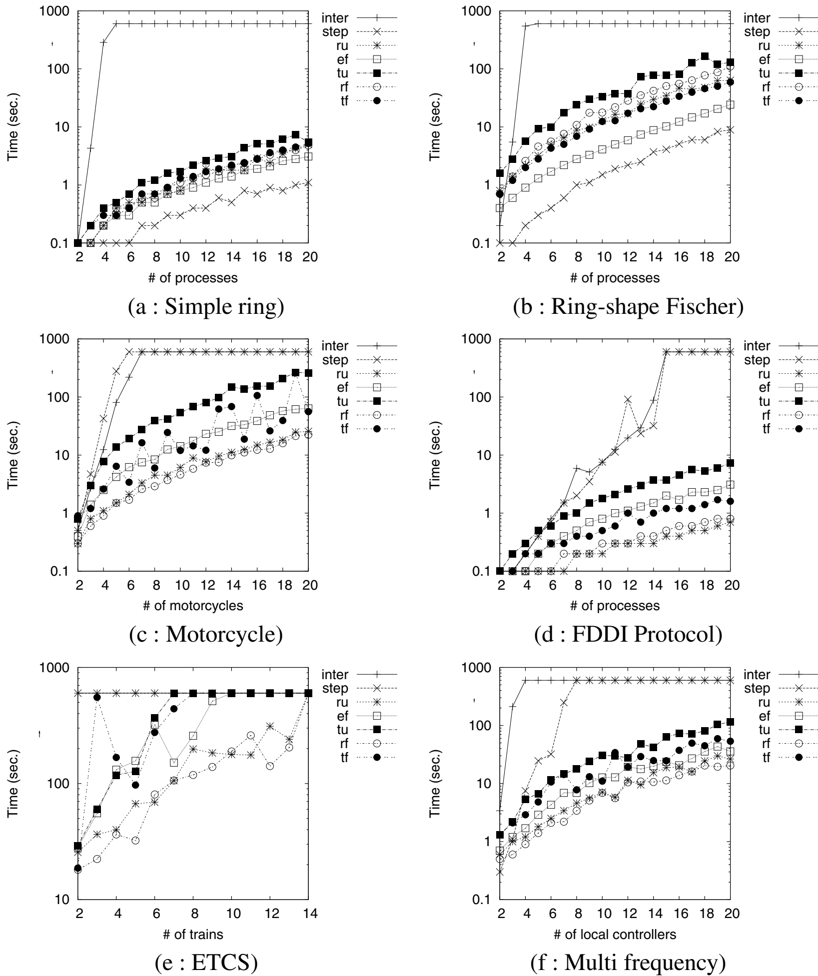


Fig. 3. Results where the length of the local runs does not depend on the number of processes

6.2 Benchmarks

We test the performance of the shallow synchronization on the following benchmarks:

- *Simple ring*: this example is a simple ring of processes where each process only communicates with its left and right neighbors; it is a proof of concept to show how the shallow synchronization can perform exponentially better than the interleaving.
- *Star-shape Fischer*: this is the hybrid fischer algorithm for the mutual exclusion protocol that uses a shared variable to control the access to a critical session.
- *Ring-shape Fischer*: this variant contains a ring of processes where each process shares a variable with its left and right neighbor; the variables are used to access critical sections in mutual exclusion with the neighbors.

- *ETCS*: this example is inspired by the European Train Control System (ETCS) specification which controls the movement of trains on a track divided into sections. The accelerated motion of the trains is approximated with linear constraints.
- *Motorcycle*: this example is inspired by the automated highway system from [14]. This system models a sequence of n motorcycles. Each motorcycle i needs to wait the signal from the previous one to move, and it needs to keep the sequence during the parade by synchronizing shared labels with neighbors.
- *FDDI Protocol*: this example is a ring topology model based on the system in [19]. It is a set of standards for data transmission on fiber optic lines in a LAN. Each component in the system waits for the signal of previous one to transmit data.
- *Nuclear Reactor*: this example from [18]. The system controls a nuclear reactor with n rods, and uses these rods to absorb neutrons one by one. Each rod that has just been moved out must stay out of the water and cool for several time units.
- *Multi-Frequency*: this example models a global controller that periodically reads the value of a variable from n local controllers, which synchronizes with an high frequency with its environment, and a lower frequency with the global controller.

6.3 Results

We check reachability problems comparing the encodings based on interleaving, step semantics, and shallow synchronization. We compared the results only on reachable instances. For unreachable cases, since we are using a BMC approach, the results strongly depend on the fixed bound, but the meaning of the bound depends on the semantics: for the interleaving, it represents the total number of local and global steps; for the shallow synchronization, it represents the maximum bound of a local run. Thus, any bound would be unfair for either semantics. Nevertheless, note that all algorithms check the unreachability of the target for path lengths smaller than the final one. So, the performance does not depend on the chance of finding the right path. We ran the experiments on a Red Hat 4.1.2 machine, with Intel(R) Core(TM)2 Quad CPU 2.66*4, and 4GB of RAM with a time out of 600 seconds.

The results of the comparison are shown in Figures 2 and 3, where the time to solve the reachability problem is plotted in log scale against the number of automata in the network. Each line corresponds to a particular option. Table 1 shows some of the features of the benchmarks, such as the length of the paths found by reachability analysis as a function of n (the number of processes in the benchmark family). Results are reported for interleaving, step semantic and shallow synchronization.

The main finding of the experimental results is that the efficiency of the bounded model checker depends on necessary depth of the search regardless the adopted semantics. The interleaving performs better than shallow synchronization in the cases where the depth of the search is the same for the different semantics (because one process interacts with all the others and its local run of one process interleaves the synchronization with all other processes): in these cases, the shallow synchronization is penalized by the overhead of the synchronizing constraints. Nevertheless, in many cases (see Fig. 3), the length of local runs do not depend on the number of processes. Thus, using the shallow semantics, we reach the target at same depth. In these cases, the encoding based on shallow synchronization scales exponentially better than the one based on interleaving. The

Table 1. Columns 2, 3 and 4 report the length of the path found with the different semantics in function of the number of processes n. Columns 5, 6, 7 report the size of the hardest instance attempted, and, in square brackets, the corresponding time, or "TO" in case of timeout. For Shallow, we report the best and worst result over the different options.

Benchmark	Path length			Hardest instance attempted		
	Inter	Step	Shallow	Inter	Step	Shallow
Simple Ring	$5n$	6	6	5[TO]	20[1.1]	20 [3.1] - 20 [5.5]
Ring-shape Fischer	$7n$	7	7	5[TO]	20[8.9]	20 [24.2] - 20 [130.2]
Star-shape Fischer	$3n$	$3n$	$3n$	8[TO]	9[TO]	5 [TO] - 6 [TO]
FDDI Protocol	$2n+1$	5	3..5	15[TO]	15[TO]	20 [0.7] - 20 [7.3]
Nuclear Reactor	$4n$	$4n$	$4n$	8[TO]	8[TO]	6 [TO] - 7 [TO]
Motorcycle	$4n+3$	$4n+3$	7..9	7[TO]	6[TO]	20 [22.4] - 20 [259.5]
ETCS	NA	NA	17	2[TO]	2[TO]	7 [TO] - 14 [TO]
Multi-Frequency	NA	$3(n-1)..3n$	9	4[TO]	8[TO]	20 [20.4] - 20 [115.6]

shorter depth of the encoding pays off the overhead due to the more complex synchronizing constraints. The same happens for the step semantics, which is the winner when it is possible to parallelize independent transitions. Among the different options of the shallow synchronization encodings, there is no winner, but using the local encoding added after reaching the target seems to win in most of cases.

We also compared our implementation with BACH, which results to be faster on many examples, while on others it does not terminate with few processes. The comparison does not help in understanding which encoding is more efficient, but rather it confirms that explicit-state search is faster on automata with a small graph, while does not compete on automata with complex graph structure. Finally, we played with different search strategies but they do not modify the outcome of the presented results. All results, together with the binaries and test cases necessary to reproduce them, are available at `http://es.fbk.eu/people/tonetta/tests/forte10/`.

7 Conclusions and Future Work

In this paper we have introduced a novel approach to symbolic reachability in networks of hybrid automata. The approach relies on the shallow synchronization semantics, that preserves the locality of reasoning within each automaton, and forces synchronization between them only when necessary. We discussed how to exploit the features of shallow synchronization in the setting of symbolic bounded model checking, exploiting some advanced features of modern SMT solvers. An experimental evaluation in the setting of linear hybrid automata shows that the proposed encodings are often more scalable than the traditional encodings based on interleaving.

In the future, we will investigate the impact of shallow synchronization to the general case of non-linear hybrid systems. Since automata synchronize only by way of discrete messages, it should be possible to integrate different reasoning engines, with different expressive power, within the same framework. The idea is to selectively apply engines to automata, and to control the search based on the computation cost associated to each

tool. Furthermore, we will investigate the application of shallow synchronization in the discrete setting, and its combination with partial order reduction techniques.

References

1. Alur, R., Dang, T., Ivancic, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embedded Comput. Syst. 5(1), 152–199 (2006)
2. Audemard, G., Bozzano, M., Cimatti, A., Sebastiani, R.: Verifying Industrial Hybrid Systems with MathSAT. Electr. Notes Theor. Comput. Sci. 119(2), 17–32 (2005)
3. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial Order Reductions for Timed Systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998)
4. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
5. Bu, L., Li, X.: Path-Oriented Bounded Reachability Analysis of Compositional Linear Hybrid Systems. Manuscript submitted (2008)
6. Bu, L., Li, Y., Wang, L., Chen, X., Li, X.: BACH2: Bounded reachAbility CHecker for Compositional Linear Hybrid Systems. In: DATE, pp. 1512–1517. EDAA (2010)
7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
8. Fränzle, M., Herde, C.: Efficient Proof Engines for Bounded Model Checking of Hybrid Systems. Electr. Notes Theor. Comput. Sci. 133, 119–137 (2005)
9. Fränzle, M., Herde, C.: HySAT: An efficient proof engine for bounded model checking of hybrid systems. Formal Methods in System Design 30(3), 179–198 (2007)
10. Giorgetti, N., Pappas, G.J., Bemporad, A.: Bounded model checking for hybrid dynamical systems. In: DAC, pp. 672–677. IEEE, Los Alamitos (2005)
11. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. In: Godefroid, P. (ed.) Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
12. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming 3(4-5), 519–550 (2003)
13. Henzinger, T.A.: The Theory of Hybrid Automata. In: LICS, pp. 278–292. IEEE Computer Society, Los Alamitos (1996)
14. Jha, S., Krogh, B., Weimer, J., Clarke, E.: Reachability for Linear Hybrid Automata Using Iterative Relaxation Abstraction. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 287–300. Springer, Heidelberg (2007)
15. Sebastiani, R.: Lazy satisability modulo theories. JSAT 3(3-4), 141–224 (2007)
16. Shinya, U.: Event order abstraction for parametric real-time system verification. In: EMSOFT, pp. 1–10. ACM, New York (2008)
17. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole Partial Order Reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008)
18. Wang, F.: Symbolic parametric safety analysis of linear hybrid systems with BDD-like data structures. IEEE Trans. Soft. Eng. 31(1), 38–51 (2005)
19. Zhao, J., Li, X., Zheng, T., Zheng, G.: Removing Irrelevant Atomic Formulas for Checking Timed Automata Efficiently. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 34–45. Springer, Heidelberg (2004)

Heap-Dependent Expressions in Separation Logic

Jan Smans, Bart Jacobs*, and Frank Piessens

Katholieke Universiteit Leuven, Belgium
{jan.smans,bart.jacobs,frank.piessens}@cs.kuleuven.be

Abstract. Separation logic is a popular specification language for imperative programs where the heap can only be mentioned through points-to assertions. However, separation logic's take on assertions does not match well with the classical view of assertions as boolean, side effect-free, potentially heap-dependent expressions from the host programming language familiar to many developers.

In this paper, we propose a variant of separation logic where side effect-free expressions from the host programming language, such as pointer dereferences and invocations of pure methods, can be used in assertions. We modify the symbolic execution-based verification algorithm used in Smallfoot to support mechanized checking of our variant of separation logic. We have implemented this algorithm in a tool and used the tool to verify some interesting programming patterns.

1 Introduction

The design of many specification languages centers around the idea that specifications should resemble the host programming language, in order to make it easy for developers to adopt and learn the specification language and to provide a straightforward semantics for run-time checking. Examples of such specification languages include the Java Modeling Language [1] and Spec# [2], where field dereferences and certain method calls can be used freely within contracts.

Over the past couple of years, separation logic [3] has proven to be a promising, powerful alternative to traditional specification formalisms. However, contrary to for instance JML and Spec#, separation logic assertions are quite different from the host programming language, in particular because the heap can only be mentioned through points-to assertions and expressions cannot mention field dereferences.

In this paper, we try to achieve the best of both worlds, by combining the power of separation logic with the programmer-friendly notation offered by traditional specification languages. In particular, we propose a variant of separation logic where side effect-free, potentially heap-dependent expressions from the host programming language can freely be mentioned inside specifications. In addition, we port the symbolic execution-based verification algorithm used in Smallfoot [4]

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 170–185, 2010.
© IFIP International Federation for Information Processing 2010

to our variant of separation logic such that program correctness can be checked mechanically.

Supporting heap-dependent expressions in separation logic is challenging for a number of reasons. First of all, as pure methods can be used in specifications, the question arises of how to frame their return values. Secondly, assertions can be ill-defined, for example because the assertion dereferences a pointer while it does not have permission to do so. In this paper, we solve the former challenge by encoding the fact that a pure method's return value depends only on values stored in the heap covered by the precondition. We solve the latter challenge by performing additional checks when assuming and producing assertions.

In summary, the contributions of this paper are as follows:

– We propose a variant of separation logic where heap-dependent expressions from the host programming language, in particular pointer dereferences and invocations of pure methods, can be used in specifications.
– We modify Smallfoot's symbolic execution-based verification algorithm to support mechanized checking of our variant of separation logic.
– We implemented our algorithm in a tool and report on experience in verifying some interesting programming patterns.

The remainder of this paper is structured as follows. Section 2 introduces our variant of separation logic and applies it in an example. In Section 3, we propose a verification algorithm for the specification language introduced in Section 2. Finally, we discuss experience with a verifier prototype, compare with related work and conclude in Sections 4, 5 and 6.

2 Separation Logic with Side Effect-Free Expressions

We describe our variant of separation logic in the context of the imperative language of Figure 1. In this figure, overlining indicates repetition; annotations are highlighted by a gray background.

A program consists of a number of declarations and a main routine $\overline{s}$. A declaration is either a mutator function, a pure function or a predicate definition. Each mutator function has a corresponding contract, consisting of a pre- and postcondition, and a method body, consisting of a number of statements followed by a return statement. Each pure function has a corresponding precondition and a method body, returning a side effect-free expression (that can potentially call the pure function itself). A predicate definition assigns a name to an assertion. A statement is either a memory allocation, a variable update, a heap update, a mutator function call, a free statement, an assert statement, an if-then-else statement, or an open or close statement. Note that the last two statements are ghost statements: they have no runtime effect and are only needed to indicate to the program verifier when folding and unfolding of predicates is required (discussed in Section 3). An expression is a variable, a constant, a pointer dereference, a pure method call, an open expression, an operator expression or an old expression. Note that all expressions are side effect-free; however some

$$
\begin{array}{lll}
\textit{program} & ::= & \overline{\textit{decl}}\ \overline{s} \\
\textit{decl} & ::= & \textit{func} \mid \textit{purefunc} \mid \textit{predicate} \\
\textit{func} & ::= & \textbf{func}\ f(\overline{x})\ \textbf{requires}\ A;\ \textbf{ensures}\ A;\ \{\ \overline{s}\ \textbf{return}\ e;\ \} \\
\textit{purefunc} & ::= & \textbf{pure}\ \textbf{func}\ p(\overline{x})\ \textbf{requires}\ A;\ \{\ \textbf{return}\ e;\ \} \\
\textit{predicate} & ::= & \textbf{predicate}\ q(\overline{x}) = A; \\
s & ::= & x := \textbf{cons}(\overline{e});\ \mid x := e;\ \mid [e] := e;\ \mid x := f(\overline{e});\ \mid \textbf{free}\ e;\ \mid \\
 & & \textbf{assert}\ e = e;\ \mid \textbf{if}(e = e)\ \{\ \overline{s}\ \}\ \textbf{else}\ \{\ \overline{s}\ \} \mid \textbf{open}\ q(\overline{e});\ \mid \textbf{close}\ q(\overline{e}); \\
e & ::= & x \mid c \mid [e] \mid p(\overline{e}) \mid \textbf{open}\ q(\overline{e})\ \textbf{in}\ e \mid e\ \textit{op}\ e \mid \textbf{old}(e) \\
A & ::= & q(\overline{e}) \mid \textbf{acc}(e) \mid A * A \mid e = e \mid \textbf{untouched}(A)
\end{array}
$$

Fig. 1. A C-like language with side effect-free, heap-dependent expressions and separation logic annotations

of them depend on the heap. An assertion is either a predicate assertion, an access assertion, a separating conjunction, an equality between expressions or an untouched assertion. An access assertion $\textbf{acc}(e)$ denotes the permission to dereference e. In classical separation logic, $\textbf{acc}(e)$ would be denoted by $e \mapsto _$. An untouched assertion $\textbf{untouched}(A)$ is a two-state assertion that holds if the values in the heap covered by A are the same in both the pre- and post-state. In the remainder of this paper, we use **true** as syntactic sugar for the assertion $0 = 0$, and **false** as syntactic sugar for $1 = 0$. In the remainder of this paper, we consider only well-formed programs (Definition 1). In our implementation, well-formedness is checked using a simple syntactic analysis.

Definition 1. *A program is* well-formed *if all of the following hold:*

- *Predicate, mutator function and pure function names are unique within the program. Parameter names are unique within a declaration.*
- *The free variables of a function's body and contract are the function parameters. Postconditions can additionally mention the variable* **result**. *The free variables of a predicate's body are the predicate's parameters. The main routine has no free variables.*
- *The program only mentions functions and predicates declared in the program text. The number of actual arguments in a function call or predicate assertion is equal to the number of formal parameters in the corresponding declaration.*
- *Old expressions and untouched assertions only appear in postconditions.*

Let us take a look at the example program of Figure 2. This program declares a predicate, a pure function and a number of mutator functions for dealing with cells, together with a main routine that uses the aforementioned functions to create and interact with cells. The pure function *get* returns the value of the cell referred to by c. Its precondition demands that c is a valid cell data structure. Note that pure functions do not have postconditions, as they cannot modify the program state. *create_cell* creates a new cell data structure with value 0.

create_cell's postcondition not only states that the result is a valid cell, but also expresses that resulting cell holds 0 via the pure function *get*. *inc* increments the value of a cell by one. Note that an old expression is used to relate the pre- and post-state. *copy* copies the value of cell d to cell c. The untouched assertion in its postcondition expresses that the heap covered by $cell(d)$ is not modified by *copy*. Client code can use the latter information to frame pure methods that depend on $cell(d)$. For example, clients can prove that $get(d)$'s return value is the same before and after calling *copy*.

The main routine creates two cells, c_1 and c_2, updates their values, checks that the value of c_1 is 1, and disposes both cells. Note that in order to prove

```
predicate cell(c) = acc(c);

pure func get(c)
   requires cell(c);
{ return open cell(c) in [c]; }

func create_cell()
   requires true; ensures cell(result) * get(result) = 0;
{ c := cons(0); close cell(c); return c; }

func inc(c)
   requires cell(c);
   ensures cell(c) * get(c) = old(get(c)) + 1;
{ open cell(c); [c] := [c] + 1; close cell(c); return 0; }

func copy(c, d)
   requires cell(c) * cell(d);
   ensures cell(c) * cell(d) * get(c) = get(d) * untouched(cell(d));
{ open cell(c); [c] := get(d); close cell(c); return 0; }

func dispose(c)
   requires cell(c); ensures true;
{ open cell(c); free(c); }

c1 := create_cell(); inc(c1);
c2 := create_cell(); inc(c2);
assert get(c1) = 1;
dispose(c1); dispose(c2);
```

Fig. 2. An annotated program written in the language of Figure 1. Annotations are highlighted by a gray background.

that the assertion never fails, one must show that creating and modifying c_2 does not affect the return value of $get(c_1)$.

The function bodies in Figure 2 contain open and close ghost statements. These statements instruct the program verifier to respectively unfold and fold predicates during symbolic execution. For example, the close statement in the body of *create_cell* removes the body of the predicate *cell* from the symbolic heap and replaces it with the predicate itself, thereby establishing the postcondition. A more detailed description of symbolic execution and open and close statements will be given in Section 3.

3 Verification

In this section, we describe the symbolic execution-based verification algorithm for our variant of separation logic. After defining how we represent program states symbolically (section 3.1), we define symbolic evaluation and execution of expressions and statements (section 3.2). Based on these definitions, we define what it means for a program to be valid (section 3.3).

3.1 Symbolic State

A symbolic state is a four-tuple (h, g, γ, π) consisting of a symbolic heap h, a symbolic pre-state heap g, a symbolic store γ and a path condition π. A symbolic heap is a multiset of heap chunks, where each heap chunk $q[s](\overline{t})$ consists of a predicate name q, a first-order term s and a list of first-order terms $\overline{t}$. We refer to the term s as the snapshot of the heap chunk. A symbolic store is a partial function from variables to first-order terms. Finally, a path condition is a set of first-order formulas, describing the conditions that hold on the current execution path.

In the remainder of this section, we describe the symbolic execution algorithm itself. The core of this algorithm consists of 4 functions: eval, produce, consume and exec. These functions respectively represent symbolic evaluation of expressions, assuming and checking assertions and symbolic execution of statements. All aforementioned functions are written in continuation passing style. That is, each function takes a continuation parameter (typically called Q) that represents the work to be done on the current path. Their signatures are as follows:

$$\begin{array}{l}
\mathsf{eval} : H \times H \times \Gamma \times \Pi \times e \times (\mathcal{T} \times \Pi \rightarrow \mathbb{B}) \rightarrow \mathbb{B} \\
\mathsf{produce} : H \times H \times \Gamma \times \Pi \times \mathcal{T} \times A \times (H \times \Pi \rightarrow \mathbb{B}) \rightarrow \mathbb{B} \\
\mathsf{consume} : H \times H \times \Gamma \times \Pi \times A \times (H \times \Pi \times \mathcal{T} \rightarrow \mathbb{B}) \rightarrow \mathbb{B} \\
\mathsf{exec} : H \times H \times \Gamma \times \Pi \times s \times (H \times \Gamma \times \Pi \rightarrow \mathbb{B}) \rightarrow \mathbb{B}
\end{array}$$

In the above signatures, H stands for the set of symbolic heaps, Γ for the set of symbolic stores, Π for the set of path conditions, $\mathcal{T}$ for the set of first-order terms and $\mathbb{B}$ for the set of booleans. Note that each of the aforementioned functions returns a boolean, indicating whether symbolic execution was successful.

3.2 Symbolic Execution

Preliminaries. The symbolic state represents symbolic values as first-order terms and information about those values as first-order formulas. In the algorithm, we use a first-order logic with equality. The signature of the logic contains a number of built-in functions, including *unit*, *pair*, *fst* and *snd*, which are used to create snapshots. A snapshot uniquely determines the values in the heap covered by a predicate. *unit* is the empty snapshot, while $pair(a, b)$ combines snapshots a and b. The functions are axiomatized as follows:

$$\forall a, b \bullet fst(pair(a, b)) = a \qquad \forall a, b \bullet snd(pair(a, b)) = b$$

We do not explicitly mention these axioms in our algorithm. Instead, we write $\pi \vdash \phi$ (denoting that formula ϕ is provable from π) as a shorthand for $\pi \cup T \vdash \phi$, where T is the theory containing the two axioms described above. Our implementation relies on the Z3 SMT solver [5] to discharge such proof tasks.

A key question our approach has to answer is how to encode pure functions during verification in a way that allows us to frame their return values. Like other verification approaches [6,7,8,9,10,11], we encode a pure function as a first-order function in the verification logic and encode a call of a pure function as an application of the corresponding first-order function. The key to solving the issue of framing is the fact that a pure function's return value can only depend on memory locations covered by its precondition. This dependence on part of the heap is encoded as an additional function parameter. As we will show in Figure 3, this parameter is the snapshot of the function's precondition. For example, the signature of the function symbol for the pure function *get* of Figure 2 is $get : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$. The first parameter represents the values in the heap covered by the precondition, while the second parameter corresponds to the pure function's parameter c.

Note that the functions produce and consume do not contain cases for $\mathbf{acc}(e)$. Instead, the algorithm considers **acc** to be just another predicate with one parameter.

If at a certain point during symbolic execution the path condition is inconsistent, then that point is not reachable during a concrete execution of the program. In our implementation, we check consistency of the path condition whenever a new formula is added to it. If adding a new formula makes the path condition inconsistent, we simply stop symbolic execution and return *true* (indicating that symbolic execution succeeded). To avoid cluttering the rules, we do not explicitly show consistency checks in the definitions of eval, consume, produce and exec.

Symbolic evaluation of expressions. $\mathsf{eval}(h, g, \gamma, \pi, e, Q)$ (defined in Figure 3) evaluates the expression e in symbolic state (h, g, γ, π) and passes both the resulting term and a potentially updated path condition to the continuation Q. More specifically, symbolic evaluation of a variable x corresponds to looking up x in the symbolic store and passing the resulting term to the continuation. A constant evaluates to itself. Dereferencing a pointer is allowed only if the thread has permission to do so. To check whether the thread has permission,

the algorithm looks in the symbolic heap for a matching chunk. If a matching chunk is found, that chunk's snapshot is passed to the continuation; otherwise, $\mathsf{eval}(h, g, \gamma, \pi, [e], Q)$ fails.

$$\mathsf{eval}(h, g, \gamma, \pi, x, Q) \equiv Q(\gamma(x), \pi)$$

$$\mathsf{eval}(h, g, \gamma, \pi, c, Q) \equiv Q(c, \pi)$$

$$\begin{array}{l}
\mathsf{eval}(h, g, \gamma, \pi, [e], Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e, (\lambda t, \pi' \bullet \\
\quad\quad \textbf{let } \mathit{matches} = \{\ \mathit{acc}[t_s](t_1) \in h \mid \pi \vdash t_1 = t\ \} \textbf{ in} \\
\quad\quad \exists \mathit{acc}[t_s](t_1) \in \mathit{matches} \bullet Q(t_s, \pi')))
\end{array}$$

$$\begin{array}{l}
\mathsf{eval}(h, g, \gamma, \pi, p(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\quad\quad \mathsf{consume}(h, g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi_n, \mathsf{precondition}(p), (\lambda h', \pi', s \bullet \\
\quad\quad\quad \mathsf{if}\ p\text{'s body visible } \mathsf{then} \\
\quad\quad\quad\quad \mathsf{produce}(h', g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi', s, \mathsf{precondition}(p), (\lambda h'', \pi'' \bullet \\
\quad\quad\quad\quad\quad \mathsf{eval}(h'', g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi'', \mathsf{body}(p), (\lambda t, \pi''' \bullet \\
\quad\quad\quad\quad\quad\quad Q(p(s, t_1, \ldots, t_n), \pi''' \cup \{p(s, t_1, \ldots, t_n) = t\}))))) \\
\quad\quad\quad \mathsf{else} \\
\quad\quad\quad\quad Q(p(s, t_1, \ldots, t_n), \pi')))))))
\end{array}$$

$$\begin{array}{l}
\mathsf{eval}(h, g, \gamma, \pi, \textbf{open } q(e_1, \ldots, e_n) \textbf{ in } e, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots (\lambda t_n, \pi_n \bullet \\
\quad\quad \mathsf{consume}(h, g, \gamma, \pi_n, q(e_1, \ldots, e_n), (\lambda h', \pi', s \bullet \\
\quad\quad\quad \mathsf{produce}(h', g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi', s, \mathsf{definition}(q), (\lambda h'', \pi'' \bullet \\
\quad\quad\quad\quad \mathsf{eval}(h', g, \gamma, \pi'', e, Q))))))))
\end{array}$$

$$\begin{array}{l}
\mathsf{eval}(h, g, \gamma, \pi, e_1\ \mathit{op}\ e_2, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\quad\quad Q(t_1\ \mathit{op}\ t_2, \pi_2)))))
\end{array}$$

$$\mathsf{eval}(h, g, \gamma, \pi, \textbf{old}(e), Q) \equiv \mathsf{eval}(g, g, \gamma, \pi, e, Q)$$

Fig. 3. Symbolic evaluation of expressions

Calling a pure function $p(e_1, \ldots, e_n)$ is allowed only if its precondition holds. Our algorithm checks whether the precondition holds by consuming it. Note that consuming the precondition not only returns an updated heap and path condition, but also a snapshot s. This snapshot is used as the first parameter in the application of the corresponding first-order function. Note that the algorithm branches on the fact whether p's body is visible. That is, if the body is visible, an assumption stating that evaluation of $p(e_1, \ldots, e_n)$ equals evaluation of its body is added to the path condition passed to the continuation. An opening expression **open** $q(e_1, \ldots, e_n)$ **in** e evaluates the expression e in a context

where the predicate $q(e_1, \ldots, e_n)$ is replaced by its body. To symbolically evaluate an binary operation $e_1\, op\ e_2$, the algorithm applies the operation to the corresponding symbolic values, t_1 and t_2. Evaluation of an old expression $\mathbf{old}(e)$ corresponds to evaluating e in the pre-state heap g.

Symbolic production of assertions. $\mathsf{produce}(h, g, \gamma, \pi, s, A, Q)$ assumes the assertion A in the symbolic state (h, g, γ, π) based on snapshot s and passes a potentially updated heap and path condition to its continuation Q. The parameter s is used to determine the snapshots of heap chunks created during production. The function $\mathsf{produce}$ is defined in terms of the helper function $\mathsf{produce}'$ as follows:

$$\mathsf{produce}(h, g, \gamma, \pi, s, A, Q) \equiv \mathsf{produce}'(\emptyset, g, \gamma, \pi, s, A, (\lambda h', \pi' \bullet Q(h \uplus h', \pi')))$$

$\mathsf{produce}'$ starts with an empty heap to ensure that assertions are self-framing. That is, the assertion A should only dereference a pointer if A itself demands access to that pointer.

Figure 4 shows the definition of $\mathsf{produce}'$. To produce a predicate assertion $q(e_1, \ldots, e_n)$, we add a predicate chunk for q to the symbolic store with snapshot s. To produce a separating conjunction $A_1 * A_2$, we must first produce A_1 and afterwards produce A_2 in the resulting symbolic state. Note that the snapshot s is split up into two pieces using the functions fst and snd. Producing an equality $e_1 = e_2$ comes down to adding the assumption that the values of both expressions are equal to the path condition. Finally, to produce $\mathbf{untouched}(A)$, we consume A in both the current symbolic heap h and the pre-state heap g, and assume that the resulting snapshots are equal.

Symbolic consumption of assertions. Consumption is the reverse of production. $\mathsf{consume}(h, g, \gamma, \pi, A, Q)$ checks if A holds in the symbolic state (h, g, γ, π) and passes a potentially updated heap, path condition and the snapshot of the

$$\begin{array}{l}
\mathsf{produce}'(h, g, \gamma, \pi, s, q(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad Q(h \uplus \{q[s](t_1, \ldots, t_n)\}, \pi_n)))))
\end{array}$$

$$\begin{array}{l}
\mathsf{produce}'(h, g, \gamma, \pi, s, A_1 * A_2, Q) \equiv \\
\quad \mathsf{produce}'(h, g, \gamma, \pi, \mathit{fst}(s), A_1, (\lambda h', \pi' \bullet \\
\qquad \mathsf{produce}'(h', g, \gamma, \pi', \mathit{snd}(s), A_2, Q)))
\end{array}$$

$$\begin{array}{l}
\mathsf{produce}'(h, g, \gamma, \pi, s, e_1 = e_2, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\qquad Q(h, \pi_2 \cup \{t_1 = t_2\})))))
\end{array}$$

$$\begin{array}{l}
\mathsf{produce}'(h, g, \gamma, \pi, s, \mathbf{untouched}(A), Q) \equiv \\
\quad \mathsf{consume}(h, g, \gamma, \pi, A, (\lambda _, _, s_1 \bullet \mathsf{consume}(g, g, \gamma, \pi, A, (\lambda _, _, s_2 \bullet \\
\qquad Q(h, \pi \cup \{s_1 = s_2\})))))
\end{array}$$

Fig. 4. Production of assertions

consumed heap chunks to the continuation Q. Note that "checking" of spatial assertions causes heap chunks to be removed from the symbolic heap. $\mathsf{consume}$ is defined in terms of the helper function $\mathsf{consume}'$ as follows:

$$\mathsf{consume}(h, g, \gamma, \pi, A, Q) \equiv \mathsf{consume}'(h, h, g, \gamma, \pi, A, Q)$$

The first symbolic heap passed to $\mathsf{consume}'$ is used for evaluating expressions, while the second represents the remainder of the original heap which is not consumed yet by the assertion.

Figure 5 shows the definition of $\mathsf{consume}'$. Consumption of a predicate assertion $q(e_1, \ldots, e_n)$ succeeds only if a heap chunk matching $q[_](t_1, \ldots, t_n)$ exists. This heap chunk is removed from the symbolic heap and its snapshot is passed to the continuation. To consume $A_1 * A_2$, one must first consume A_1 and afterwards consume A_2. A pair containing the snapshots of A_1 and A_2 is passed to the continuation. Consumption of $e_1 = e_2$ succeeds only if both expressions are provably (from the path condition) equal and the continuation Q succeeds. Finally, consumption of **untouched**(A) succeeds only if the snapshots obtained by consuming A in both the pre- and post-state are provably equal.

$$\begin{array}{l}
\mathsf{consume}'(h, h', g, \gamma, \pi, q(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\qquad \mathbf{let}\ \mathit{matches} = \{q[s](t'_1, \ldots, t'_1) \in h' \mid \pi_n \vdash t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n\}\ \mathbf{in} \\
\qquad \exists q[s](t'_1, \ldots, t'_1) \in \mathit{matches} \bullet Q(h' - \{q[s](t'_1, \ldots, t'_1)\}, \pi_n, s)))))
\end{array}$$

$$\begin{array}{l}
\mathsf{consume}'(h, h', g, \gamma, \pi, A_1 * A_2, Q) \equiv \\
\quad \mathsf{consume}'(h, h', g, \gamma, \pi, A_1, (\lambda h'', \pi', s_1 \bullet \\
\qquad \mathsf{consume}'(h, h'', g, \gamma, \pi', A_2, (\lambda h''', \pi'', s_2 \bullet Q(h''', \pi'', \mathit{pair}(s_1, s_2))))))
\end{array}$$

$$\begin{array}{l}
\mathsf{consume}'(h, h', g, \gamma, \pi, e_1 = e_2, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\qquad (\pi_2 \vdash t_1 = t_2) \wedge Q(h', \pi_2)))))
\end{array}$$

$$\begin{array}{l}
\mathsf{consume}'(h, h', g, \gamma, \pi, s, \mathbf{untouched}(A), Q) \equiv \\
\quad \mathsf{consume}'(h, h, h, \gamma, \pi, A, (\lambda _, _, s_1 \bullet \mathsf{consume}'(g, g, g, \gamma, \pi, A, (\lambda _, _, s_2 \bullet \\
\qquad\quad (\pi \vdash s_1 = s_2) \wedge Q(h', \pi)))))
\end{array}$$

Fig. 5. Consumption of assertions

Symbolic execution of statements. $\mathsf{exec}(h, g, \gamma, \pi, s, Q)$ (Figure 6) symbolically executes statement s in symbolic state (h, g, γ, π) and passes a potentially updated heap, store and path condition to the continuation Q. More specifically, a memory allocation $x := \mathbf{cons}(e_1, \ldots, e_n);$ is modeled by creating a fresh term representing the address of the newly allocated memory. The heap is extended with n new memory locations starting at address l containing the terms t_1 to t_n. The variable x is modified to l. A variable update $x := e;$ modifies the value of the variable x in the symbolic store γ to the symbolic value of

e. A heap update $[e_1] = e_2$; is allowed only if the symbolic heap contains a chunk that matches $acc[_](t_1)$. If such a match exists, that heap chunk's snapshot is changed to t_2; otherwise, symbolic execution fails. Symbolic execution of a mutator call $x := f(e_1, \ldots, e_n)$; consists of consuming f's precondition and producing f's postcondition afterwards. Note that a fresh term is used as the snapshot for producing the postcondition. A free statement **free** e; is allowed only if the heap contains a chunk matching $acc[_](t)$. If it does, then that chunk is removed from the heap; otherwise, symbolic execution fails. Note that the chunks produced by $\mathbf{cons}(e_1, \ldots, e_n)$ need to be freed separately. As assert statement **assert** $e_1 = e_2$; fails if the values of e_1 and e_2 are not provably equal; otherwise, the assert statement is equivalent to skip. An if-then-else statement $\mathbf{if}(e_1 = e_2)$ $\{\ \overline{s_1}\ \}$ **else** $\{\ \overline{s_2}\ \}$ splits symbolic execution into two branches. An open statement **open** $q(e_1, \ldots, e_n)$; removes a chunk matching $q[_](t_1, \ldots, t_n)$ from the symbolic state and produces q's body with the snapshot of the removed chunk; if no such chunk exists, symbolic execution fails. Finally, a close statement **close** $q(e_1, \ldots, e_n)$; consumes q's body and replaces the consumed chunks with a predicate chunk for q. The snapshot obtained by consuming q's body is used as the snapshot of this new heap chunk.

3.3 Valid Program

We say that a mutator function is valid (Definition 2) if after producing the precondition for arbitrary values of the parameters and executing the mutator's body in the resulting state, consumption of the postcondition succeeds and the symbolic heap is empty. A pure function is valid (Definition 3) if evaluation of its body does not fail in the symbolic state resulting from producing the precondition for arbitrary values of the parameters. A predicate is valid (Definition 4) if production of its body does not fail for arbitrary parameters. Finally, the main routine is valid (Definition 5) if its symbolic execution succeeds in an empty heap, pre-state heap, store and path condition and the resulting heap is empty.

A program is valid if all functions and the main routine are valid. The program of Figure 2 is valid.

Definition 2. *A mutator function*

$$\mathbf{func}\ f(x_1, \ldots, x_n)\ \mathbf{requires}\ A_1;\ \mathbf{ensures}\ A_2;\ \{\ \overline{s};\ \mathbf{return}\ e;\ \}$$

is valid if the following holds:

$$\begin{array}{l}
\mathbf{let}\ (t_1, \ldots, t_n) = (\mathsf{fresh}, \ldots, \mathsf{fresh})\ \mathbf{in} \\
\mathsf{produce}(\emptyset, \emptyset, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \emptyset, \mathsf{fresh}, A_1, (\lambda h, \pi \bullet \\
\quad \mathsf{exec}(h, h, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi, \overline{s}, (\lambda h', \gamma, \pi' \bullet \\
\quad\quad \mathsf{eval}(h', h, \gamma, \pi', e, (\lambda t, \pi'' \bullet \\
\quad\quad\quad \mathsf{consume}(h', h, \{(x_1, t_1), \ldots, (x_n, t_n), (\mathbf{result}, t)\}, \pi'', A_2, \\
\quad\quad\quad\quad (\lambda h'', _, _ \bullet h'' = \emptyset))))))))
\end{array}$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, x := \mathbf{cons}(e_1, \ldots, e_n);, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\quad\quad \mathbf{let}\ l = \mathsf{fresh}\ \mathbf{in} \\
\quad\quad Q(h \uplus \{acc[t_1](l), \ldots acc[t_n](l+n-1)\}, \gamma[x \mapsto l], \pi_n)))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, x := e;, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e, (\lambda t, \pi_1 \bullet Q(h, \gamma[x \mapsto t], \pi_1)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, [e_1] := e_2;, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\quad\quad \mathbf{let}\ matches = \{acc[s](t'_1) \mid \pi_2 \vdash t_1 = t'_1\}\ \mathbf{in} \\
\quad\quad \exists acc[s](t'_1) \in matches \bullet Q(h - \{acc[s](t'_1)\} \uplus \{acc[t_2](t_1)\}, \gamma, \pi_2)))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, x := f(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\quad\quad \mathsf{consume}(h, g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi_n, \mathsf{precondition}(f), (\lambda h', \pi', _ \bullet \\
\quad\quad\quad \mathbf{let}\ (s, r) = (\mathsf{fresh}, \mathsf{fresh})\ \mathbf{in} \\
\quad\quad\quad \mathsf{produce}(h', h, \{(x_1, t_1), \ldots, (x_n, t_n), (\mathbf{result}, r)\}, \pi', s, \mathsf{postcondition}(f), (\lambda h'', \pi'' \bullet \\
\quad\quad\quad\quad\quad Q(h'', \gamma[x \mapsto r], \pi''))))))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{free}\ e;, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e, (\lambda t, \pi' \bullet \\
\quad\quad \mathbf{let}\ matches = \{acc[s](t') \mid \pi' \vdash t = t'\}\ \mathbf{in} \\
\quad\quad \exists acc[s](t') \in matches \bullet Q(h - \{acc[s](t')\}, \gamma, \pi')))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{assert}\ e_1 = e_2;, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\quad\quad (\pi_2 \vdash t_1 = t_2) \wedge Q(h, \gamma, \pi_2)))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{if}(e_1 = e_2)\ \{\ \overline{s_1}\ \}\ \mathbf{else}\ \{\ \overline{s_2}\ \}, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \mathsf{eval}(h, g, \gamma, \pi_1, e_2, (\lambda t_2, \pi_2 \bullet \\
\quad\quad \mathsf{exec}(h, g, \gamma, \pi_2 \cup \{t_1 = t_2\}, \overline{s_1}, Q) \wedge \mathsf{exec}(h, g, \gamma, \pi_2 \cup \{t_1 \neq t_2\}, \overline{s_2}, Q)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{open}\ q(e_1, \ldots, e_n);, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\quad\quad \mathsf{consume}(h, g, \gamma, \pi_n, q(e_1, \ldots, e_n), (\lambda h', \pi', s \bullet \\
\quad\quad\quad \mathsf{produce}(h', g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi, s, \mathsf{definition}(q), (\lambda h'', \pi'' \bullet \\
\quad\quad\quad\quad Q(h'', \gamma, \pi''))))))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, \mathbf{close}\ q(e_1, \ldots, e_n);, Q) \equiv \\
\quad \mathsf{eval}(h, g, \gamma, \pi, e_1, (\lambda t_1, \pi_1 \bullet \ldots \mathsf{eval}(h, g, \gamma, \pi_{n-1}, e_n, (\lambda t_n, \pi_n \bullet \\
\quad\quad \mathsf{consume}(h, g, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi_n, \mathsf{definition}(q), (\lambda h', \pi', s \bullet \\
\quad\quad\quad Q(h' \uplus \{q[s](t_1, \ldots t_n)\}, \gamma, \pi')))))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exec}(h, g, \gamma, \pi, s_0\ \overline{s}, Q) \equiv \\
\quad \mathsf{exec}(h, g, \gamma, \pi, s_0, (\lambda h', \gamma', \pi' \bullet \mathsf{exec}(h', g, \gamma', \pi', \overline{s}, Q)))
\end{array}
$$

Fig. 6. Symbolic execution of statements

Definition 3. *A pure function*

$$\textbf{pure func } p(x_1, \ldots, x_n) \textbf{ requires } A; \ \{ \textbf{ return } e; \ \}$$

is valid if the following holds:

$$\begin{array}{l} \textbf{let } (t_1, \ldots, t_n) = (\mathsf{fresh}, \ldots, \mathsf{fresh}) \textbf{ in} \\ \mathsf{produce}(\emptyset, \emptyset, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \emptyset, \mathsf{fresh}, A, (\lambda h, \pi \bullet \\ \quad \mathsf{eval}(h, h, \{(x_1, t_1), \ldots, (x_n, t_n)\}, \pi, e, (\lambda _, _ \bullet \mathit{true})))) \end{array}$$

Definition 4. *A predicate*

$$\textbf{predicate } q(x_1, \ldots, x_n) = A;$$

is valid if the following holds:

$$\mathsf{produce}(\emptyset, \emptyset, \{(x_1, \mathsf{fresh}), \ldots, (x_n, \mathsf{fresh})\}, \emptyset, \mathsf{fresh}, A, (\lambda h, \pi \bullet \mathit{true}))$$

Definition 5. *A main routine* $\overline{s}$ *is valid if the following holds:*

$$\mathsf{exec}(\emptyset, \emptyset, \emptyset, \emptyset, \overline{s}, (\lambda h, _, _ \bullet h = \emptyset))$$

Pure Method Termination. It is essential for the soundness of our approach that pure methods terminate. Verification therefore includes a phase that checks sufficient conditions for pure method termination. Specifically, it is checked for each pure method call in a pure method body that either (1) the callee is defined earlier in the program text, or (2) the call is in the body of an **open** expression, or (3) there is some symbolic heap chunk that is not consumed by the precondition of the call. This ensures that at each call, either the size of the symbolic heap decreases, or the *derivation depth* decreases (i.e. the number of close operations required to construct the heap from one that contains only field chunks), or the position in the program text decreases. Since the size and the derivation depth are always finite and a pure method cannot increase the size or the derivation depth of the symbolic heap, this ensures termination. Contrary to pure functions, mutator functions are not required to terminate.

4 Implementation and Experience

We have implemented the algorithm described in Section 3 in a tool. The source code (F#), binaries and a number of examples are available from the author's website http://www.cs.kuleuven.be/~jans/speccheck. Instead of the C-like language used in the paper, the tool supports a larger assertion language (e.g. conditional assertions) for a small subset of C#. To check whether a first-order formula is derivable from the path condition, we use the Z3 SMT solver [5]. Our verifier prototype has been used to verify a number of small programming patterns, including aggregate objects and iterator. These programs together with their verification times are shown in Table 1. To help developers diagnose verification errors, our verifier includes a symbolic debugger (shown in Figure 7). When verification fails, the developer can inspect the components of the symbolic states encountered during symbolic execution on the path to the failure.

Table 1. Programs verified using the verifier prototype together with the verification time (seconds). The experiments were executed on a standard desktop machine with a 2.66 Ghz processor and 4 GB of RAM running Windows Vista. cell50 is the similar to the main routine of Figure 2, except that 50 intermediate cells are created and updated instead of 1.

example	cell	abstract cell	cell50	interval	iterator
time taken (seconds)	0.005	0.008	0.18	0.03	0.01

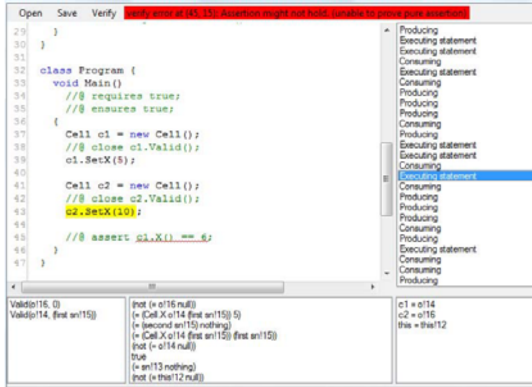


Fig. 7. A screenshot of the verifier prototype. Developers can use the symbolic debugger in the IDE to diagnose verification errors and inspect the symbolic state at each program point. The box on the right of the screen contains a list of symbolic states encountered during symbolic execution. The boxes at the bottom of the screen show the symbolic state (from left to right: the symbolic heap, the path condition and the symbolic store) at a particular program point.

5 Related Work

Separation logic [3] extends Hoare logic with three new assertions: **emp**, separating conjunction and points-to. Our assertion **acc**(e) is similar to separation logic's points-to assertion $e \mapsto _$, which denotes that the current thread has permission to access the memory at address e (without constraining the value at that address). The key difference between classical separation logic and our variant is that we allow expressions that are used inside assertions to depend on the heap. Examples of heap-dependent expressions include pointer dereferences and function invocations. Note that we do not claim that our variant has additional expressive power. Instead, the difference leads to a different style of writing specifications. More specifically, where we use pure methods to express the state of an object, classical separation logic relies on predicate parameters. For example, the contract of *inc* of Figure 2 would be written in separation logic as: **requires** $cell(c, X)$; **ensures** $cell(c, X + 1)$;. Here, X is a logical variable.

Berdine *et al.* [4] have proposed a symbolic execution-based verification algorithm for programs annotated with separation logic specifications and have implemented this algorithm for a small imperative language in Smallfoot. The algorithm described in Section 3 is a variant of the aforementioned algorithm. In particular, the idea of dividing the symbolic state in spatial and pure assertions (the symbolic heap and the path condition respectively) and the rules dealing with heap access, non-heap-dependent expressions and assertions are largely similar to those in [4]. The novel aspect of our approach is the treatment of pure methods and pointer dereferences inside assertions via snapshots. Smallfoot only supports a limited number of predicates, but the developer does not need to write open and close statements as the tool has hard-coded, built-in rules for reasoning about those predicates.

jStar [12] and VeriFast [13] extend the basic ideas of Berdine *et al.* to full-fledged programming languages like Java and C. While loop invariants must be provided by the developer in our approach, jStar infers certain loop invariants automatically, provided the developer inputs the necessary abstraction rules. Using an SMT solver [5] for discharging pure queries and supporting symbolic debugging via an IDE that shows the components of the symbolic state are ideas taken from VeriFast.

As an alternative to the Smallfoot's symbolic execution-based verification algorithm, Leino and Müller [14] and Smans *et al.* [15] have proposed an approach based on verification condition generation and automated theorem proving for a variant of separation logic. However, experience has shown that approaches based on verification condition generation and automated theorem proving in general, and [14,15] in particular, have 3 disadvantages: (1) they are slow, (2) they are unpredictable, as small changes in the specification can have a significant impact on verification time and (3) verification errors are hard to diagnose, as it is hard to determine whether the specification is flawed or whether the theorem prover is unable to prove a particular part of the verification condition. As the experiments with our verifier prototype indicate, verification times are consistently low. For example, even if we increase the number of intermediate statements in the main routine of Figure 2, the verification time only increases marginally. The main reason why verification is fast is that we reason about the heap outside of the theorem prover, and hence do not need to send quantifier-heavy formulas (in particular quantifiers about the heap) to the automated prover. These experiments thus confirm earlier results with similar algorithms [4,12,13]. Moreover, the developer can diagnose verification errors by inspecting the symbolic states on the path to the failure. A disadvantage of the approach presented in this paper with respect to [14,15] is that non-separating conjunction is not supported.

Reasoning about method calls in specifications and framing their return values in particular was posed as a challenge for verification by Leavens, Leino and Müller [16]. In the context of approaches based on verification condition generation and automated theorem proving, researchers have attacked well-formedness of pure method specifications [6,7], framing of return values [8,9,10] and allowing certain side effects in pure methods [11,9]. The approach proposed in this paper

is similar to existing techniques in the sense that we also encode pure methods as functions and invocations of pure methods as function applications. Moreover, the idea of using snapshots is similar to the snapshots used in [8,9]. However, to the best of our knowledge, this is the first paper that discusses the use of pure methods and framing of their return values in the context of separation logic (and in the context of its symbolic execution-based verification algorithm).

6 Conclusion

In this paper, we combined the expressive power of separation logic with the programmer-friendly notation of specification languages such as JML where heap-dependent expressions can be used in annotations. We proposed an algorithm to support mechanized checking for this variant of separation logic and implemented this algorithm in a tool.

References

1. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. Behavioral Specifications of Businesses and Systems (1999)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
5. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Rudich, A., Darvas, Á., Müller, P.: Checking well-formedness of pure-method specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 68–83. Springer, Heidelberg (2008)
7. Leino, K.R.M., Middelkoop, R.: Practical reasoning about invocations and implementations of pure methods. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503. Springer, Heidelberg (2009)
8. Jacobs, B., Piessens, F.: Inspector methods for state abstraction. Journal of Object Technology 6(5) (2007)
9. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 336–351. Springer, Heidelberg (2007)
10. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: Automatic verification of java programs with dynamic frames. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, Springer, Heidelberg (2008)
11. Darvas, Á., Müller, P.: Reasoning about method calls in interface specifications. Journal of Object Technology 5(5) (2006)
12. Distefano, D., Parkinson, M.: jStar: Towards practical verification for Java. In: OOPSLA (2008)

13. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven (2008)
14. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
15. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009 – Object-Oriented Programming. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
16. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. In: Schmorrow, D.D., Reeves, L.M. (eds.) HCII 2007 and FAC 2007. LNCS (LNAI), vol. 4565. Springer, Heidelberg (2007)

Static Type Analysis of Pattern Matching by Abstract Interpretation

Pietro Ferrara

ETH Zurich, Switzerland
pietro.ferrara@inf.ethz.ch

Abstract. Pattern matching is one of the most attractive features of functional programming languages. Recently, pattern matching has been applied to programming languages supporting the main current object oriented features. In this paper, we present a static type analysis based on the abstract interpretation framework aimed at proving the exhaustiveness of pattern matchings and the safety of type casts. The analysis is composed by two distinct abstract domains. The first domain collects information about dynamic typing, while the second one tracks the types that an object cannot be instance of. The analysis has been implemented and applied to all the Scala library. The experimental results underline that our approach scales up since it analyzes a method in 90 msec in average. In addition, the analysis is precise in practice as well, since we prove the exhaustiveness of 42% of pattern matchings and 27% of the type casts without any manual annotation on the code.

Keywords: Abstract interpretation, static analysis, pattern matching.

1 Introduction

Pattern matching is recognized to be one of the most expressive features of functional programming languages. Extending its full expressiveness to programming languages supporting the current object oriented features is not straightforward [12]. In fact, features like inheritance and information hiding are not supported by the functional pattern matching, since it dealt with algebraic data types. Nevertheless, recent work [14,26] introduced pattern matching in programming languages supporting the main current object oriented features as F# [1] and Scala [23]. In addition, some extensions of Java supporting pattern matching appeared in the last few years [16,24]. Pattern matching checks if an expression respects a given pattern looking at an ordered list of case expressions. One of the most common case expressions is the type case: we select the case to apply following the dynamic type of the expression. Consider the Scala program in Figure 1. The content of argument x is matched with respect to its type: if it is an instance of Something, the method returns the string representing the contained value. Otherwise, the type of x is None, and extract returns the empty string.

Languages like F# and Scala are compiled into a bytecode language (for instance, MSIL [11] or Java bytecode [18]) that supports the main imperative

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 186–200, 2010.
© IFIP International Federation for Information Processing 2010

features. Thus type cases in pattern matching are translated into type tests and casts. For instance, the body of method extract in Figure 1 is translated into the code in Figure 2 by the Scala compiler [13]. Note that information about generics is erased.

```
abstract sealed class Option[T]
 final case class Something[T](val y : T) extends Option[T]
 final case object None extends Option[Nothing]

def extract [T](x : Option[T]) : String = x match {
  case Something(y) => return y.toString();
  case None => return "";
}
```

Fig. 1. Pattern matching with type cases

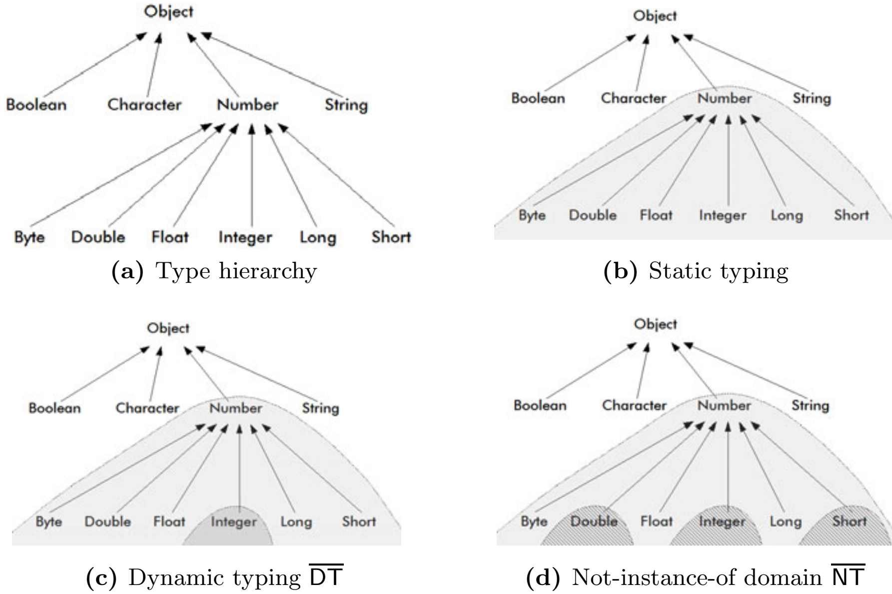
```
1 String extract (Option x) {
2     if (x instanceOf Something)
3         return ((Something) x).y. toString ();
4     else if (x instanceOf None)
5             return "";
6         else throw new MatchError ();
7 }
```

Fig. 2. Results of Scala compilation

This code can be easily optimized adopting a specific static analysis. When x instanceof Something is false, we know that x instanceof None is always true. In fact, the static type of x is Option, this class is abstract, and the only two classes that extend it are Something and None. Note that Option is declared as sealed. In Scala, a sealed class can be extended only by classes that are defined in the same source file. Therefore we know that Option cannot be extended by external code, and we can conclude that the if statement at line 4 is always evaluated to true. In addition, we know that the statement throw new MatchError() is unreachable, that is the pattern matching contained by the original Scala program is exhaustive. Finally, the type cast at line 3 is safe.

1.1 Contribution

The main contribution of this paper is the introduction of two new abstract domains in order to capture precise information on the types of variables. The two main goals of our analysis are precision and efficiency. Precision is achieved through the development of abstract domains focused on the properties of interest, that is, exhaustiveness of pattern matching and safety of casts. Efficiency is achieved through a modular analysis. For this reason, our approach is based on



(a) Type hierarchy

(b) Static typing

(c) Dynamic typing $\overline{\mathsf{DT}}$

(d) Not-instance-of domain $\overline{\mathsf{NT}}$

Fig. 3. Type system and abstract domains

the Design-by-Contract (DbC) methodology [21]. This means that when we analyze a method call we rely on class invariants, pre and post conditions. Thanks to this approach, we do not need to analyze the whole program, but we can analyze a method alone relying on its contracts. In a similar way, we rely on language constructs like sealed in Scala in order to be sure that a class cannot be extended by external code.

In this paper, we do not deal with issues related to the heap and its analysis. Hence the language on which we are going to define our analysis does not contain field accesses, aliasing, etc., but it deals only with variables. The object oriented programs we analyzed are preprocessed by a sound heap analysis that replaces heap accesses with variables. This means that method calls may have side effects on these variables. The heap analysis we perform is quite rough: intuitively, we approximate all the instances of the same class with the same variable, and we perform weak updates [4] as these abstract references may represent several concrete references. Then the number of abstract references is bounded to the number of classes of the analyzed program. In this way we preserve the scalability of our approach. On the other hand, it may seem that this approach is too much coarse since we approximate all the fields having the same static type with the same variable. Indeed, this does not particularly affect the precision of our approach because, for the most part of the programs we analyzed, the content of fields is first stored in a local variable (and in this way we obtain a specific variable identifier), and then type checking and casts are performed on this local variable.

Consider now the type hierarchy depicted in Figure 3a. The class Object is extended by several subclasses that represent different types in a programming language, e.g., Boolean. The abstract class Number is extended by several subclasses, each representing one possible numerical type, e.g., Integer. For instance, Figure 3b represents a situation in which the static type of a variable is Number. The static type information guarantees that the objects assigned to a variable are instances of its static type or of one of its subclasses. Thus instances of Long, Float, etc. may be assigned to a variable whose static type is Number.

The first abstract domain we are going to define is $\overline{\mathsf{DT}}$. It is aimed at approximating information about the dynamic type of variables. Figure 3c depicts a case in which we assign an object instance of the Integer class (darker area) to a variable of static type Number (light grey area). The information about dynamic typing comes mainly from assignments, type casts, new statement, and assumption of instanceof conditions. Since we rely on DbC, if a method has no postcondition about the dynamic type of variables (e.g., the returned value), we would abstract its type information using its static type after the method call.

The second domain we are going to define is $\overline{\mathsf{NT}}$. It abstracts the types that a variable cannot be instance of. For instance, Figure 3d depicts a situation in which a variable of static type Number cannot be instance of Double, Integer, or Short. In this way we can discard some leaves of the type tree. This information is collected when we assume the negation of an instanceof boolean condition. We could have adopted a different approach capturing a set of types for each variable representing all the possible types a variable can have at a given program point. In this way, we may model that an object cannot be instance of a type simply removing the type from the set of types related to it. Unluckily this solution is not feasible for two main reasons. Imagine to be after a method call, where we know only the static type of a variable. In this context, assuming that an object cannot be instance of a given type (i) may affect the performance (e.g., if its static type is Object and we have to exclude String we should relate the variable to a set of types of huge dimensions), (ii) cannot be represented since our analysis is modular and external code usually defines and instantiates new classes. One may argue that, since we are interested in proving the exhaustiveness of pattern matching, we can restrict the analysis only on types that cannot be extended by external code. With such restriction the approach based on set of possible types would be feasible, and performance could be preserved supposing that we have few subclasses. First of all, we want to track information also on classes that can be externally extended, in order to eventually provide information to developers like "if this class would be sealed, then the pattern matching would be exhaustive". This solution would not collect the information necessary to provide this output. In addition, we do not expect in general that sealed classes (or more generally classes that cannot be extended by external code) have only few subclasses, since each subclass may be used to represent a particular case of a complex data structure.

Combining $\overline{\mathsf{DT}}$ and $\overline{\mathsf{NT}}$, we are in position to obtain precise information about types and prove the safety of some cast operations, and the unreachability of some code, in particular when dealing with pattern matching.

The analysis works on a simple object oriented language. We designed it in order to translate the main object oriented programming languages into this language. In this way, we want to apply our analysis to several languages. Up to now, we have developed the translation of the Scala programming language. Thus our analysis is focused on the code obtained from the Scala pattern matching statements. In this context, if we prove that a MatchError exception is unreachable, this means that the pattern matching is exhaustive.

The analysis has been implemented and the experimental results underline that our approach scales up. We are in position to analyze Scala libraries (more than 30.000 methods) in less than one hour, and in average we require 90 milliseconds to analyze a single method. In addition, it turns out that the analysis is precise in practice. In fact, we are able to prove the exhaustiveness of 42% of pattern matchings (that may contain not only type cases but also other cases whose information is not captured by our analysis) and 27% of the type casts without any manual annotation of the code.

The following of this section introduces some preliminaries on abstract interpretation. Section 2 presents the language that we analyze. Section 3 introduces the concrete domain and semantics, while Section 4 formalizes the two domains adopted by our analysis, and explains how they work on an example. Section 5 presents the experimental results, and Section 6 introduces the related work. Finally, Section 7 concludes and discusses the future work.

1.2 Abstract Interpretation

Our approach is based on abstract interpretation [7,8], a theory for defining and soundly approximating the semantics of a program. A concrete semantics, aimed at specifying the runtime properties of interest, is defined. Then it is approximated to obtain an abstract semantics computable but still precise enough to capture the property of interest, and specified by an abstract domain and an abstract transition function.

Formally, the concrete domain D is a lattice $\langle \mathsf{D}, \sqsubseteq_\mathsf{C}, \bot_\mathsf{C}, \top_\mathsf{C}, \sqcup_\mathsf{C}, \sqcap_\mathsf{C}\rangle$. The concrete elements are related to the abstract domain $\langle \overline{\mathsf{D}}, \sqsubseteq_\mathsf{A}, \bot_\mathsf{A}, \top_\mathsf{A}, \sqcup_\mathsf{A}, \sqcap_\mathsf{A}\rangle$ by a concretization function γ and an abstraction function α. In order to obtain a sound analysis, we require that they form a Galois connection; we denote it by $\langle \mathsf{D}, \sqsubseteq_\mathsf{C}, \bot_\mathsf{C}, \top_\mathsf{C}, \sqcup_\mathsf{C}, \sqcap_\mathsf{C}\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \overline{\mathsf{D}}, \sqsubseteq_\mathsf{A}, \bot_\mathsf{A}, \top_\mathsf{A}, \sqcup_\mathsf{A}, \sqcap_\mathsf{A}\rangle$.

A semantics is defined on the abstract and concrete domains. Given a state of the domain and a statement, it produces the result of the execution of the statement starting from the given state. The abstract semantics $\overline{\mathbb{S}}$ has to soundly approximate the concrete one, that is $\forall \overline{\mathsf{d}} \in \overline{\mathsf{D}} : \mathbb{S}[\![\gamma(\overline{\mathsf{d}})]\!] \sqsubseteq_\mathsf{A} \gamma \circ \overline{\mathbb{S}}[\![\overline{\mathsf{d}}]\!]$. Usually, the semantics is defined as the computation of a fixpoint [8].

Different domains can be combined in Cartesian products. Let $\overline{\mathsf{A}} \times \overline{\mathsf{B}}$ be the Cartesian product of abstract domains $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$. The lattice operators are defined as the pointwise application on the components of the pairs of the lattice operator

Table 1. OO core language

C ::=	x = E	(C1)	E ::=	new T()	(E1)	B ::= x instanceof T	(B1)
	declare x : T	(C2)		x	(E2)	! x instanceof T	(B2)
	assume(B)	(C3)		(T) x	(E3)		
	x.M(y1, ..., yn)	(C4)					

on each abstract domain. The reduced product $\overline{\mathsf{A}} \otimes \overline{\mathsf{B}}$ is a Cartesian product on which a *reduce* function is provided: given a state of the Cartesian product, it mutually refines the information contained in each domain using the information provided by the other domain.

Generic analyzers. Abstract interpretation can be applied in order to develop generic analyzers [5]. In particular, this theory allows one to define a compositional analysis, e.g., an analysis that can be instantiated with different numerical domains, and in order to analyze different properties. Many different generic analyzers have been proposed recently [15,19,25]. The type analysis we present in this paper has been implemented in Sample (Static Analyzer of Multiple Programming LanguagEs). This generic analyzer can be plugged with different numerical domains and heap analysis. In addition, it can be extended with new analyses like our type analysis.

2 Language

We introduce a core language in order to make explicit the main characteristics of our abstract domains. It consists of assignments (C1), declarations of variables (C2), assumptions of boolean conditions (C3), and method calls (C4). An expression that can be assigned to a variable is the instantiation of a class (E1), another variable (E2), or the cast of a variable (E3). A condition can be the check that the dynamic type is instance of (B1) or it is not instance of (B2) a type.

We focus our attention only on the main aspects of the type analysis we are interested in, that is, casts and runtime checks of dynamic types. Nevertheless, our implementation covers all the features of current programming languages, as numerical operations, field accesses, etc. Intuitively, we represent the body of a method through a Control Flow Graph (CFG). Each block in the CFG contains a list of statements. Different blocks are connected through edges that could eventually contain a boolean condition to represent conditional jumps. In this way, we support control structures like if statements and while loops.

3 Concrete Domain and Collecting Semantics

The concrete domain is composed by environments that collect the dynamic type for each variable. Formally, $\mathsf{Env} : [\mathsf{Var} \rightarrow \overline{\mathsf{T}}]$. The concrete domain is composed

Table 2. Collecting semantics

$\mathbb{E}$	$\mathbb{B}$	$\mathbb{S}$
$\mathbb{E}[\![\mathsf{new}\ \mathsf{T}(), \mathsf{d}]\!] = \mathsf{T}$	$\mathbb{B}[\![\mathsf{x}\ \mathsf{instanceof}\ \mathsf{T}, \mathsf{d}]\!] = \mathsf{d}(\mathsf{x}) \sqsubseteq_{\overline{\mathsf{T}}} \mathsf{T}$	$\mathbb{S}[\![\mathsf{x} = \mathsf{E}, \mathsf{d}]\!] = \mathsf{d}[\mathsf{x} \mapsto \mathbb{E}[\![\mathsf{E}, \mathsf{d}]\!]]$
$\mathbb{E}[\![\mathsf{x}, \mathsf{d}]\!] = \mathsf{d}(\mathsf{x})$	$\mathbb{B}[\![!\mathsf{B}, \mathsf{d}]\!] = \neg \mathbb{B}[\![\mathsf{B}, \mathsf{d}]\!]$	$\mathbb{S}[\![\mathsf{declare}\ \mathsf{x} : \mathsf{T}, \mathsf{d}]\!] = \mathsf{d}[\mathsf{x} \mapsto \mathsf{T}]$
$\mathbb{E}[\![(\mathsf{T})\ \mathsf{x}, \mathsf{d}]\!] = \begin{cases} \mathsf{d}(\mathsf{x}) & \text{if } \mathsf{d}(\mathsf{x}) \sqsubseteq_{\overline{\mathsf{T}}} \mathsf{T} \\ \bot & \text{otherwise} \end{cases}$		$\mathbb{S}[\![\mathsf{assume}(\mathsf{B}), \mathsf{d}]\!] = \begin{cases} \bot & \text{if } \mathbb{B}[\![\mathsf{B}, \mathsf{d}]\!] = \mathsf{false} \\ \mathsf{d} & \text{if } \mathbb{B}[\![\mathsf{B}, \mathsf{d}]\!] = \mathsf{true} \end{cases}$
		$\mathbb{S}[\![\mathsf{x.M}(\mathsf{y1}, ..., \mathsf{yn}), \mathsf{d}]\!] = \mathit{execute}(\mathsf{x.M}(\mathsf{y1}, ..., \mathsf{yn}), \mathsf{d})$

by sets of environments. Each environment represents one possible execution of the program in a given point. The lattice is obtained using set operators. Formally, $\langle \wp(\mathsf{Env}), \subseteq, \cup, \cap, \mathsf{Env}, \emptyset \rangle$.

We define the collecting semantics as a function that, given an environment and a statement, returns the environment resulting from the execution of the given statement on the given environment. Table 2 reports the formal definition of this collecting semantics[1]. $\sqsubseteq_{\overline{\mathsf{T}}}$ corresponds to the subtype relation, while $\bot$ is used to represent a situation in which the execution is stopped because of an unsafe dynamic cast. The collecting semantics is aimed at formalizing the runtime behaviors focusing on the information we are interested. The extension of this semantics to our concrete domain (that is made by set of environments) is the application of this semantics on each environment in the initial state.

$\mathbb{S}$ is the basic step adopted in order to compute the semantics of a method on its CFG representation. Intuitively, we build up incrementally the traces representing the executions of a method relying on $\mathbb{S}$ to perform single computational steps [6]. Since non-deterministic behaviors are possible (e.g., because of inputs from the user) and these may also affect the type information, we consider sets of traces.

4 Abstract Domain and Semantics

In this section, we present and formalize the two abstract domains of our analysis, how we combine them, and we explain how they work in practice on the example presented in Figure 2. Note that we will define the abstract semantics only on single statements. Its extension to blocks and CFGs is obtained as usual in approaches based on abstract interpretation and trace semantics relying on the upper bound operators [7].

4.1 Static Typing

The programming language introduced in Section 2 provides some information on the static types of variables. We suppose that a subtype relation $\sqsubseteq_{\overline{\mathsf{T}}}$ is provided, and that this is a partial ordering. Given two types $\bar{\mathsf{t}}_1, \bar{\mathsf{t}}_2 \in \overline{\mathsf{T}}$, $\bar{\mathsf{t}}_1 \sqsubseteq_{\overline{\mathsf{T}}} \bar{\mathsf{t}}_2$ if

[1] $\mathit{execute}(\mathsf{x.M}(\mathsf{y1}, ..., \mathsf{yn}), \mathsf{d})$ resolves and executes method M on d.

and only if $\bar{\mathsf{t}}_1$ is subtype of $\bar{\mathsf{t}}_2$. We denote by $\sqcup_{\overline{\mathsf{T}}}$ and $\sqcap_{\overline{\mathsf{T}}}$ respectively the upper and lower bound operators univocally identified by $\sqsubseteq_{\overline{\mathsf{T}}}$. These operators form a lattice. In this way, we support the most part of common object oriented type systems [3], like the ones of Java, C#, Scala, and F#.

Each type represents itself and all the types that are its subtype. Note that this is exactly the semantics of a static type: at runtime a variable of static type $\bar{\mathsf{t}}$ can have a type that is $\bar{\mathsf{t}}$ or one of its subtypes. Formally, the concretization of a type $\bar{\mathsf{t}}$ is defined as $\gamma_{\overline{\mathsf{T}}}(\bar{\mathsf{t}}) = \{\overline{\mathsf{t}'} : \overline{\mathsf{t}'} \sqsubseteq_{\overline{\mathsf{T}}} \bar{\mathsf{t}}\}$. Since information about static typing can be considered as syntactic sugar, we suppose that a function *statictype* : $[\mathsf{Var} \rightarrow \overline{\mathsf{T}}]$ is provided. Given a variable, it returns its static type.

4.2 $\overline{\mathsf{DT}}$: Dynamic Typing

$\overline{\mathsf{DT}}$ abstracts the dynamic type of variables. Relying on $\overline{\mathsf{T}}$ we build up the domain $\overline{\mathsf{DT}} : [\mathsf{Var} \rightarrow \overline{\mathsf{T}}]$ that relates each variable to its dynamic type. The operators on the lattice $\langle \overline{\mathsf{DT}}, \sqsubseteq_{\overline{\mathsf{DT}}}, \sqcup_{\overline{\mathsf{DT}}}, \sqcap_{\overline{\mathsf{DT}}}, \top_{\overline{\mathsf{DT}}}, \bot_{\overline{\mathsf{DT}}} \rangle$ are obtained as the functional extension of the ones of $\overline{\mathsf{T}}$. The information inferred by $\overline{\mathsf{DT}}$ is refined with the static type information provided by the *statictype* function: if the type of a variable v is not yet defined, the dynamic type domain returns its static type, that is, *statictype*(v). The concretization of $\overline{\mathsf{DT}}$ is defined as the functional extension of $\gamma_{\overline{\mathsf{T}}}$ as well. Formally, $\gamma_{\overline{\mathsf{DT}}}(\bar{\mathsf{f}}) = \{[\mathsf{v} \mapsto \bar{\mathsf{t}}] : \mathsf{v} \in dom(\bar{\mathsf{f}}) \wedge \bar{\mathsf{t}} \in \gamma_{\overline{\mathsf{T}}}(\bar{\mathsf{f}}(\mathsf{v}))\}$.

Semantics

$$\overline{\mathbb{DT}}[\![\mathsf{x} = \mathsf{y}, \overline{\mathsf{dt}}]\!] = \overline{\mathsf{dt}}[\mathsf{x} \mapsto \overline{\mathsf{dt}}(\mathsf{y})]$$
$$\overline{\mathbb{DT}}[\![\mathsf{x} = \mathsf{new\ T}(), \overline{\mathsf{dt}}]\!] = \overline{\mathsf{dt}}[\mathsf{x} \mapsto \mathsf{T}]$$
$$\overline{\mathbb{DT}}[\![\mathsf{x} = (\mathsf{T})\mathsf{y}, \overline{\mathsf{dt}}]\!] = \overline{\mathsf{dt}}[\mathsf{x} \mapsto \mathsf{T} \sqcap_{\overline{\mathsf{T}}} \overline{\mathsf{dt}}(\mathsf{y}), \mathsf{y} \mapsto \mathsf{T} \sqcap_{\overline{\mathsf{T}}} \overline{\mathsf{dt}}(\mathsf{y})]$$
$$\overline{\mathbb{DT}}[\![\mathsf{assume}(\mathsf{x\ instanceof\ T}), \overline{\mathsf{dt}}]\!] = \overline{\mathsf{dt}}[\mathsf{x} \mapsto \overline{\mathsf{dt}}(\mathsf{x}) \sqcap_{\overline{\mathsf{T}}} \mathsf{T}]$$
$$\overline{\mathbb{DT}}[\![\mathsf{x.M}(\mathsf{y1}, ..., \mathsf{yn}), \overline{\mathsf{dt}}]\!] = \overline{\mathbb{DT}}[\![\mathsf{assume}(\mathsf{rename}(\mathsf{postcondition}_{\mathsf{M}})), \top_{\overline{\mathsf{DT}}}]\!]$$

When a variable is assigned to another one ($\mathsf{x} = \mathsf{y}$), we capture that the dynamic type of x is the type of y. Similarly, when we assign to a variable the instance of a fresh object, we relate this variable to the type of the new object. The dynamic type of the assignment of a cast to type T of a variable y is the lower bound between the dynamic type of y and type T. Thus, it may be the case in which it contains a type that is more approximated than T even if the cast is safe. After the cast, we know that its dynamic type will be surely T or one of its subtypes. This is why we take the lower bound between the dynamic type of the variable and T. In a similar way, when we test to true a boolean condition like x instanceof T, we track that the dynamic type of x is the lower bound of its type and T. When we analyze a method call, we assume the postconditions of this method call, that may contain some type information, on the top state of the $\overline{\mathsf{DT}}$ domain. Since we do not consider framing information, we suppose that potentially a method may access and assign all the variables of the current state of computation. Then we assume the postcondition on a top state in order to preserve the soundness of our analysis. In addition, we suppose that a

rename function is provided, and it renames all the variables contained in the postcondition matching the calling environment.

Running Example. At the beginning of the analysis of method extract we have no information about dynamic typing, but we know that the static type of x is Option. When we analyze the condition of the if branch at line 2 we calculate that the state leading to the then statement of our dynamic type domain is $[\mathsf{x} \mapsto \mathsf{Something}]$, since the condition is evaluated to true. In the same way, at line 4 we obtain that $[\mathsf{x} \mapsto \mathsf{None}]$. Thus we prove that the cast at line 3 is safe, while we are not able yet to detect that line 6 is unreachable.

4.3 $\overline{\mathsf{NT}}$: Not-instance-of Domain

The second domain we introduce is $\overline{\mathsf{NT}}$. This domain is aimed at collecting the types an object cannot be instance of at a given point of the program.

First of all, we define a lattice that collects sets of types. The intuition behind them is that they contain all the types of which a variable cannot be instance of. Thus let $\overline{\mathsf{NT}}$ be this domain, i.e., $\overline{\mathsf{NT}} = \wp(\overline{\mathsf{T}})$. Its concretization contains all the types that are not in the given set or that are subtypes of one of the types in the set. Formally, $\gamma_{\overline{\mathsf{NT}}}(\overline{\mathsf{f}}) = \{[\mathsf{v} \mapsto \overline{\mathsf{t}} : \mathsf{v} \in dom(\overline{\mathsf{f}}) \wedge \nexists \overline{\mathsf{t}'} \in \overline{\mathsf{f}}(\mathsf{v}) : \overline{\mathsf{t}} \sqsubseteq_{\overline{\mathsf{T}}} \overline{\mathsf{t}'}]\}$. The ordering operator on $\overline{\mathsf{NT}}$ is not the subset operator, as more types we have, more precise we are. On the other hand, it is neither the superset operator, as several types may be subtypes of the same type, and they are different elements of the set. Thus the ordering operator is defined as

$$\overline{\mathsf{nt}_1} \sqsubseteq_{\overline{\mathsf{NT}}} \overline{\mathsf{nt}_2} \Leftrightarrow \forall \mathsf{v} \in dom(\overline{\mathsf{nt}_2}) : \mathsf{v} \in dom(\overline{\mathsf{nt}_1}) \wedge \forall \overline{\mathsf{t}_2} \in \overline{\mathsf{nt}_2}(\mathsf{v}) : \exists \overline{\mathsf{t}_1} \in \overline{\mathsf{nt}_1}(\mathsf{v}) : \overline{\mathsf{t}_2} \sqsubseteq_{\overline{\mathsf{T}}} \overline{\mathsf{t}_1}$$

The other lattice operators are the ones induced by $\sqsubseteq_{\overline{\mathsf{NT}}}$. The bottom element for a single variable is the set containing only the top type (e.g., Object in Java or Any in Scala). The concretization of this element would be all the types that are not subtype of the top type, i.e., the empty set. We denote by $\bot_{\overline{\mathsf{NT}}}$ a function that relates all the variables to $\{\top_{\overline{\mathsf{T}}}\}$. Formally, $\bot_{\overline{\mathsf{NT}}} = \lambda \mathsf{x}.\{\top_{\overline{\mathsf{T}}}\}$. The top element is the empty function. Formally, $\langle \overline{\mathsf{NT}}, \sqsubseteq_{\overline{\mathsf{NT}}}, \sqcup_{\overline{\mathsf{NT}}}, \sqcap_{\overline{\mathsf{NT}}}, \bot_{\overline{\mathsf{NT}}}, \emptyset \rangle$.

Semantics

$$
\begin{array}{l}
\overline{\mathbb{NT}}[\![\mathsf{x} = \mathsf{y}, \overline{\mathsf{nt}}]\!] = \overline{\mathsf{nt}}[\mathsf{x} \mapsto \overline{\mathsf{nt}}(\mathsf{y})] \\
\overline{\mathbb{NT}}[\![\mathsf{x} = \mathsf{new\ T}, \overline{\mathsf{nt}}]\!] = \overline{\mathsf{nt}}[\mathsf{x} \mapsto \emptyset] \\
\overline{\mathbb{NT}}[\![\mathsf{x} = (\mathsf{T})\mathsf{y}, \overline{\mathsf{nt}}]\!] = \overline{\mathsf{nt}}[\mathsf{x} \mapsto \overline{\mathsf{nt}}(\mathsf{y})] \\
\overline{\mathbb{NT}}[\![\mathsf{assume}(!\ \mathsf{x\ instanceof\ T}, \overline{\mathsf{nt}})]\!] = \overline{\mathsf{nt}}[\mathsf{x} \mapsto \overline{\mathsf{nt}}(\mathsf{x}) \cup \{\mathsf{T}\}] \\
\overline{\mathbb{NT}}[\![\mathsf{x.M}(\mathsf{y1}, ..., \mathsf{yn}), \overline{\mathsf{dt}}]\!] = \overline{\mathbb{DT}}[\![\mathsf{assume}(\mathsf{reaname}(\mathsf{postcondition}_{\mathsf{M}})), \top_{\overline{\mathsf{NT}}}]\!]
\end{array}
$$

We are interested in collecting which types a variable is not instance of. When a variable y is assigned to x (with or without cast), we track that the types x cannot be instance of are the same of y. When we assign a new instance of a

class to a variable, we forget everything we knew about that variable. In this way, we lose some information: we could consider that the variable cannot be instance of all the types that are not subtype of the instantiated class. On the other hand, this would be computationally expensive, the same information is already contained in $\overline{\mathsf{DT}}$, and we are going to combine this domain with $\overline{\mathsf{NT}}$. $\overline{\mathsf{NT}}$ is also interested to approximate information when a boolean condition like x instanceof List is tested to false. For instance, when we analyze the if statement if(x instanceof List) C1; else C2, we know that in the else branch x cannot be instance of List, and we add List to the state of the $\overline{\mathsf{NT}}$ domain before analyzing C2. The method call is managed as in $\overline{\mathsf{DT}}$: method calls may have side effects on variables, arbitrarily changing their dynamic type. In order to preserve the soundness of our approach, the only information we have after the method call is that the postcondition of the called method holds.

Running Example. The $\overline{\mathsf{NT}}$ domain models information on the example presented in Figure 2 when we test to false conditions containing instanceof. Thus at line 4 (i.e., the else branch of the if statement at line 2) its state is $[\mathsf{x} \mapsto \{\mathsf{Something}\}]$. At line 6 (i.e., the else branch of the if statement at line 4) its state is $[\mathsf{x} \mapsto \{\mathsf{Something}, \mathsf{None}\}]$. Unluckily, $\overline{\mathsf{NT}}$ alone is not enough in order to prove that line 6 is unreachable, as it knows nothing about the static or dynamic types of x.

4.4 $\overline{\mathsf{FT}}$: Reduced Product of $\overline{\mathsf{DT}}$ and $\overline{\mathsf{NT}}$

The $\overline{\mathsf{DT}}$ domain infers the dynamic type of a variable. $\overline{\mathsf{NT}}$ collects the types a variable cannot be instance of. Thus these domains model different types of information, but we can mutually refine them. In particular, we are interested in checking if the information contained by $\overline{\mathsf{NT}}$ about a variable is not compatible with the one contained in $\overline{\mathsf{DT}}$. In this case, we know that this point of the program is unreachable, and thus we can refine our domain to the bottom state. For instance, consider an abstract sealed class I (thus it cannot be instantiated nor extended by external code) that is implemented by two classes O1 and O2. If we know from $\overline{\mathsf{DT}}$ that the dynamic type of a variable x is I, but from $\overline{\mathsf{NT}}$ we know that x cannot be instance of O1 nor O2, then there is no possible type for x.

Formally, let be $\overline{\mathsf{FT}}$ the reduced product of $\overline{\mathsf{DT}}$ and $\overline{\mathsf{NT}}$, i.e., $\overline{\mathsf{FT}} = \overline{\mathsf{DT}} \otimes \overline{\mathsf{NT}}$. The lattice operators of $\langle \overline{\mathsf{FT}}, \sqsubseteq_{\overline{\mathsf{FT}}}, \sqcup_{\overline{\mathsf{FT}}}, \sqcap_{\overline{\mathsf{FT}}}, \top_{\overline{\mathsf{FT}}}, \bot_{\overline{\mathsf{FT}}} \rangle$ are defined as usual when dealing with the product of domains. The reduction function $\overline{red}_{\overline{\mathsf{FT}}}$ is defined as

$$\overline{red}_{\overline{\mathsf{FT}}} : [\overline{\mathsf{FT}} \rightarrow \overline{\mathsf{FT}}]$$

$$\overline{red}_{\overline{\mathsf{FT}}}((\overline{\mathsf{d}}, \overline{\mathsf{n}})) = \begin{cases} \bot_{\overline{\mathsf{FT}}} & \text{if } \exists \mathsf{x} \in dom(\overline{\mathsf{d}}) : \overline{getType}(\mathsf{x}, \overline{\mathsf{d}}) = \bar{\mathsf{t}} \wedge \\ & \forall \overline{\mathsf{t}_1} \in \gamma_{\overline{\mathsf{T}}}(\bar{\mathsf{t}}) : \overline{abstract}(\overline{\mathsf{t}_1}) = \mathsf{false} \wedge \exists \overline{\mathsf{t}_2} \in \overline{\mathsf{n}} : \overline{\mathsf{t}_1} \sqsubseteq_{\overline{\mathsf{T}}} \overline{\mathsf{t}_2} \wedge \\ & \qquad \bar{\mathsf{t}} \text{ cannot be extended by external code} \\ (\overline{\mathsf{d}}, \overline{\mathsf{n}}) & \text{otherwise} \end{cases}$$

Table 3. Results of the analysis of the example

Line	$\overline{\mathsf{DT}}$	$\overline{\mathsf{NT}}$	Reduced $\overline{\mathsf{FT}}$
2	$[\mathsf{x} \mapsto \mathsf{Option}]$	$\emptyset$	
3	$[\mathsf{x} \mapsto \mathsf{Something}]$	$\emptyset$	
4	$[\mathsf{x} \mapsto \mathsf{Option}]$	$[\mathsf{x} \mapsto \{\mathsf{Something}\}]$	
5	$[\mathsf{x} \mapsto \mathsf{None}]$	$[\mathsf{x} \mapsto \{\mathsf{Something}\}]$	
6	$[\mathsf{x} \mapsto \mathsf{Option}]$	$[\mathsf{x} \mapsto \{\mathsf{Something}, \mathsf{None}\}]$	$\bot_{\overline{\mathsf{FT}}}$

where $\overline{abstract} : [\overline{\mathsf{T}} \rightarrow \{\mathsf{true}, \mathsf{false}\}]$ is a function that, given a type, checks if the given type cannot be instantiated, e.g., it is an interface or an abstract class in Java, or a trait in Scala.

The reduction function is aimed at discovering if the information contained in $\overline{\mathsf{NT}}$ is not compatible with the information contained in $\overline{\mathsf{DT}}$. It returns $\bot_{\overline{\mathsf{FT}}}$ if and only if (i) all the types a variable can be instance of following the information contained in $\overline{\mathsf{DT}}$ are excluded at least by one type contained in $\overline{\mathsf{NT}}$, (ii) the type related to the variable in $\overline{\mathsf{DT}}$ cannot be extended by external code (e.g., it is declared as sealed in Scala). In this way we achieve the modularity in our analysis. Note that this reduction is partial, as there may be other ways to refine the information contained in the different domains. Anyway, we did not find useful other reductions of the information contained in this domain in order to improve the precision of our analysis.

The concretization of $\overline{\mathsf{FT}}$ is defined as the intersection of the pointwise application of $\gamma_{\overline{\mathsf{DT}}}$ and $\gamma_{\overline{\mathsf{NT}}}$. Formally, $\gamma_{\overline{\mathsf{FT}}}((\overline{\mathsf{d}}, \overline{\mathsf{n}})) = \gamma_{\overline{\mathsf{DT}}}(\overline{\mathsf{d}}) \cap \gamma_{\overline{\mathsf{NT}}}(\overline{\mathsf{n}})$. The semantics $\overline{\mathbb{FT}}$ is defined as the pointwise application of $\overline{\mathbb{DT}}$ and $\overline{\mathbb{NT}}$.

Theorem 1. *The fixpoint trace semantics based on $\overline{\mathbb{FT}}$ is sound with respect to the one based on $\mathbb{S}$.*

Running Example. Table 3 reports the results of the $\overline{\mathsf{FT}}$ domain when applied to method extract of the example presented in Figure 2. We already pointed out how the information is captured by the abstract domains. The reduced product refines the information contained in the two domains at line 6 discovering that it is not possible for variable x to be instance of Option, and not to be instance of Something and None at the same time. In fact, these two classes represent all the possible instances of Option. Note that Option is declared as sealed, hence it cannot be extended by external code. Therefore the reduced product is able to discover that line 6 is unreachable, and MatchError cannot be thrown. Previously, we discovered that $\overline{\mathsf{DT}}$ proved that the cast at line 3 is safe, thus, in a Java environment, a ClassCastException cannot be thrown. Using this information, we may improve the runtime performance (for instance removing the last if statement since the else branch is unreachable) and provide useful information to developers in order to debug programs, since we guarantee that pattern matching is exhaustive in the original Scala code.

5 Experimental Results

				Casts			MatchErrors		
	#m	**t**	**tm**	**v**	**nv**	**p**	**v**	**nv**	**p**
actors	1626	3'56"	145.35	47	104	31.13%	20	17	54.05%
collection	14578	10'21"	42.63	183	509	26.45%	42	74	36.21%
util	4926	8'21"	101.75	126	508	10.87%	36	65	35.64%
xml	2786	6'21"	136.59	108	286	27.41%	5	21	19.23%
mainlib	8218	12'02"	87.82	97	196	33.11%	37	12	75.51%
scala lib	35732	54'02"	90.72	725	1951	27.09%	156	208	42.86%

We applied our analysis to all the Scala library v. 2.7.7. We executed the analysis on an Intel Code 2 Quad CPU 2.83 GHz with 4 GB of RAM, running Windows 7, and the Java SE Runtime Environment 1.6.0_16-b01. The preceding table reports the experimental results. Column **#m** reports the number of analyzed methods. Column **t** reports the time required to analyze these methods[2], while **tm** reports the time required in average to analyze a single method in milliseconds. We report the number of validated cases (column **v**), not validated (**nv**) cases, and the overall precision (**p**) of our analysis when analyzing both type casts and reachability of MatchError exceptions.

The Scala library contains more than 35.000 methods. In average we analyze a method in 90 milliseconds. Thus our analysis is quite efficient, and it scales up. When we analyze the type casts, we are not able to distinguish between casts due to the compilation of pattern matchings, and other casts, since our analysis works on the code obtained after the transformations and simplifications performed by the Scala compiler. Hence our analysis takes into account all the casts contained in the program. We are able to automatically prove safe more than 27% of all the casts. We think that this result goes beyond our initial goal (that was, to precisely analyze pattern matching), since we proved the safety of more than 700 casts while there were only 364 MatchError exceptions (that means that there were 364 pattern matchings). On the other hand, in general a precision less than 30% is not particularly satisfying. We analyzed these warnings, and it turns out that the most part is due to erasure of type generics of Scala compiler. For instance, when we access the next element of a list of Integer objects, this is compiled into a program that takes the next element of a list of Object instances and casts it to Integer. Obviously, since we do not have information about generics, we cannot prove the safety of this code. We plan to apply our analysis to languages that preserve generic type information in order to study how this feature can improve the precision of our analysis. A minor part of the warnings is induced by the lack of contracts. Since the Scala library is not annotated, each time we have a method call we forget everything is contained in $\overline{\mathsf{DT}}$ and $\overline{\mathsf{NT}}$. We expect the we may improve the precision of our analysis adding some contracts concerning framing information in particular.

[2] This time takes into account also the heap analysis.

About the reachability of MatchError, we proved that about 43% of these exceptions is unreachable. This result is particular encouraging: even if we took into account a part of the information that can be used in pattern matching and we do not annotate the code with contracts, we proved that almost half of the existing pattern matchings is safe, and we can remove the MatchError exception. Usually we were not able to prove the exhaustiveness of pattern matchings when they contain some checks on numerical information. In a minor part of the cases, we failed to prove the exhaustiveness because the pattern matching was not exhaustive since it expected to receive a particular type of expression. Contracts would be the solution in order to express that a method expects that an argument can have only some specific types (e.g., not all the ones that are subtype of its static type), thus we expect that we could improve the precision adding this annotation.

6 Related Work

The related work is addressed mainly into two directions: the static analysis of type information in order to optimize virtual calls, and static analysis of pattern matching.

The first topic has been studied during the last 15 years. Object oriented languages introduced the idea of virtual calls: we do not know at compile time exactly which method we are calling, but we need some runtime information (for instance, the dynamic type of the object on which we are calling the method). Thus the binding of the called method is dynamic, and this step may require an overhead. Applying static analysis to reduce and bound the number of virtual calls improves the runtime performance of programs. In this context, many approaches were proposed [2]. Some of them performs an inter-procedural analysis [17], while our approach relies on contracts. Other approaches analyze the class hierarchy statically [9], and check if the virtual call can refer only to one implementation of the method in this context. These approaches usually consider information only about dynamic types.

A recent work [20] proposed a type analysis based on the abstract interpretation theory in order to optimize JavaScript code. This analysis is focused on the numerical information that can be assigned to a variable in order to infer if we can adopt an Int32 type instead of a Float64 type. Using this information the code is optimized. This approach is focused on the numerical information and adopts this information to infer the types of variables, while our approach approximates information on the types of variables to check which casts are safe and if some statements are unreachable.

Recent work applied static analysis techniques to prove properties of pattern matching. Mitchell and Runciman [22] proposed an analysis to check if non-exhaustive pattern matching in Haskell could lead to failures. In particular, it infers preconditions that are strong enough to prove that the pattern matching never fails. Our approach is aimed at discovering which pattern matchings are non-exhaustive, instead of inferring which constraints are necessary in order to make exhaustive a pattern matching that is not.

Dotta *et al.* [10] introduced a verification system that checks the disjointness, reachability, and exhaustiveness of Scala pattern matching. Global information about the program is inferred through different kinds of formulas. The authors do not formalize how these formulas are inferred, and there is no proof about the soundness of the approach. Instead, we fully formalized our approach and we proved its soundness. On the other hand, their analysis tracks more information than ours. Their implementation requires up to some seconds to prove the exhaustiveness of some case studies. Our analysis tracks information only of type-based pattern matchings, but it is scales up.

7 Conclusion and Future Work

In this paper we presented a static type analysis focused on pattern matching. We introduced and combined two distinct abstract domains, each abstracting different information (dynamic typing, and not-instance-of information). We proved the soundness of our approach relying on the abstract interpretation framework. The analysis has been implemented in Sample, and the experimental results proved the scalability and the precision of our approach.

As future work, we plan to combine our analysis with some numerical domains in order to study if considering this information improves the precision of our analysis. In addition, we plan to exploit the information captured by our analysis to optimize the programs resulting from the Scala compiler, and to study how much this optimization may improve the runtime performance of some Scala benchmarks.

References

1. F#, http://research.microsoft.com/fsharp
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA 1996, pp. 324–341. ACM, New York (1996)
3. Cardelli, L.: Type systems. In: Tucker, A.B. (ed.) The Computer Science and Engineering Handbook, ch. 97. CRC Press, Boca Raton (2004)
4. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI 1990. ACM, New York (1990)
5. Cousot, P.: The calculational design of a generic abstract interpreter. In: Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
6. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science 277, 47–103 (2002)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977. ACM, New York (1977)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979. ACM, New York (1979)
9. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)

10. Dotta, M., Suter, P., Kuncak, V.: On static analysis for expressive pattern matching. Technical report, EPFL (2008)
11. Standard ECMA-335. Common Language Infrastructure (CLI), 4th edn. ECMA (June 2006)
12. Emir, B.: Object-oriented pattern matching. PhD thesis, EPFL (2007)
13. Emir, B., Ma, Q., Odersky, M.: Translation correctness for first-order object-oriented pattern matching. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 54–70. Springer, Heidelberg (2007)
14. Emir, B., Odersky, M., Williams, J.: Matching objects with patterns. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 273–298. Springer, Heidelberg (2007)
15. Ferrara, P.: Checkmate: a generic static analyzer of java multithreaded programs. In: Proceedings of SEFM 2009. IEEE Computer Society, Los Alamitos (2009)
16. Hirzel, M., Nystrom, N., Bloom, B., Vitek, J.: Matchete: Paths through the pattern matching jungle. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 150–166. Springer, Heidelberg (2008)
17. Kumar, R., Chakraborty, S.S.: Precise static type analysis for object oriented programs. SIGPLAN Not. 42, 17–26 (2007)
18. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
19. Logozzo, F., Fähndrich, M.: On the relative completeness of bytecode analysis versus source code analysis. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 197–212. Springer, Heidelberg (2008)
20. Logozzo, F., Venter, H.: Rata: Rapid atomic type analysis by abstract interpretation. Application to JavaScript optimization. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 66–83. Springer, Heidelberg (2010)
21. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall, Englewood Cliffs (1997)
22. Mitchell, N., Runciman, C.: Not all patterns, but enough: An automatic verifier for partial but sufficient pattern matching. SIGPLAN Not. 44, 49–60 (2009)
23. Odersky, M.: The Scala Language Specification (2008)
24. Richard, A., Lhotak, O.: Oomatch: pattern matching as dispatch in Java. In: OOPSLA 2007, pp. 771–772. ACM, New York (2007)
25. Spoto, F.: Julia: A Generic Static Analyser for the Java Bytecode. In: Proceedings of FTfJP 2005 (2005)
26. Syme, D., Neverov, G., Margetson, J.: Extensible pattern matching via a lightweight language extension. SIGPLAN Not. 42(9), 29–40 (2007)

On-the-Fly Trace Generation and Textual Trace Analysis and Their Applications to the Analysis of Cryptographic Protocols

Yongyuth Permpoontanalarp

Logic and Security Laboratory
Department of Computer Engineering, Faculty of Engineering
King Mongkut's University of Technology Thonburi, Bangkok, Thailand
yongyuth.per@kmutt.ac.th

Abstract. Many model checking methods have been developed and applied to analyze cryptographic protocols. Most of them can analyze only one attack trace of a found attack. In this paper, we propose a very simple but practical model checking methodology for the analysis of cryptographic protocols. Our methodology offers an efficient analysis of all attack traces for each found attack, and is independent to model checking tools. It contains two novel techniques which are on-the-fly trace generation and textual trace analysis. In addition, we apply our method to two case studies which are TMN authenticated key exchanged protocol and Micali's contract signing protocol. Surprisingly, it turns out that our simple method is very efficient when the numbers of traces and states are large. Also, we found many new attacks in those protocols.

Keywords: Formal methods for cryptographic protocols, Model checking, Cryptographic protocols.

1 Introduction

Cryptographic protocols are protocols which use cryptographic techniques to achieve certain tasks while preventing malicious parties to attack the protocols. There are many applications of cryptographic protocols, for example, authenticated key exchange protocols, web security protocols, e-payment protocols, e-banking protocols, e-voting protocols, etc.

The design and analysis of cryptographic protocols are difficult to achieve because of the increasingly attacking capabilities and the complex requirement of the applications. Attacks in many cryptographic protocols have been found later after they have been designed [1, 2] and even implemented eg. [3, 4]. Thus, it requires a method to analyze all possible attacks to the protocols. Such kind of method would offer a comprehensive understanding of all vulnerabilities of protocols and certainly would help in developing a better protection for them. Note that in this paper we focus on only message replay attacks [5].

Many model checking methods [6-16] have been developed and applied to analyze cryptographic protocols. Most of them can analyze only one attack trace of a found

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 201–215, 2010.
© IFIP International Federation for Information Processing 2010

attack. In fact, all of them employ *off-the-fly* trace generation technique. It means that after a state space is generated either partially or fully and an attack state is found, an attack trace is then computed. This kind of trace generation is called *off-the-fly* since the trace computation occurs after the state space is generated. An attack trace is constructed by searching for a path from an initial state to an attack state. Since the searching for all paths between two states is extremely time-consuming, only one path is searched, instead, in most methods. However, the analysis of single attack trace is rather limited since one path or one attack trace represents only one way amongst many possible ways to carry out an attack. In addition, after one attack trace is found, a *visualization* technique is normally employed to illustrate and analyze the attack, for example, message sequence charts [31] and graphs. Such visualization technique can provide an intuitive analysis of a single attack trace. However, when the number of traces is large, for example thousands, it is hard to analyze them by visualizing, eg. to classify them into groups.

In this paper, we propose a very simple but practical model checking methodology for the analysis of cryptographic protocols. Our methodology offers an efficient analysis of all attack traces for each found attack, and is independent to model checking tools. The study of all attack traces is beneficial since it offers a deep understanding on all attackers' capabilities to compromise a system. Our method contains two novel techniques which are *on-the-fly* trace generation and *textual* trace analysis. In our method, while a state space is generated, attack traces for states are computed at the same time and stored at the states themselves. We call it *on-the-fly* trace generation since the trace computation occurs at the same time as the state space computation. Thus, after the whole state space is computed and an attack is found, then attack traces for the attack can be extracted from attack states of the attack immediately. Thus, all attack traces can be computed very efficiently without any path searching. This technique provides a big improvement in the computation time for all attack traces when the number of attack traces and the number of states are large.

The number of attack traces obtained can be quite large. For example, we found 1,020 traces for an attack in the TMN protocol. So, we propose *textual* trace analysis technique to classify such large number of attack traces. Those attack traces are classified by using attack patterns. Attack patterns are minimal but necessary protocol traces for an attack. Attack traces that contain the same attack pattern are classified into the same group of attack traces. While the development of an attack pattern is manual, the attack classification is automatic. By using our two new techniques, protocol designers could obtain a deep and thorough analysis of all possible attacks to cryptographic protocols.

To demonstrate the practical uses of our approach, we apply our new methodology to two case studies which are Micali's contract signing protocol (ECS1) [17] and TMN authenticated key exchange protocol [18]. We implement our methodology in a model checker tool called CPNTools [19,20]. Note that CPNTools originally provides the *off-the-fly* trace generation only. Then, we compare the results between our *on-the-fly* trace generation and the *off-the-fly* trace generation both in CPNTools. Surprisingly, it turns out that our *on-the-fly* trace generation is more efficient than the *off-the-fly* trace generation when the numbers of traces and states are large. For TMN, our result shows that when the numbers of states and traces are 74,244 and 13,056, respectively, our method improves on the computation times for 6,777 %. For ECS1,

our result shows that when the numbers of states and traces are 235,564 and 7,032, respectively, our method improves on the computation times for 116.75 %.

Because our method can analyze all attack traces, we found many new attacks in the two protocols in our previous works [21-24]. For Micali's contract signing protocol, we found one new single-session attack [21] and two new multi-session attacks [22]. Also, we found three new attacks [22] of Bao et. al.'s modified version of ECS1 [27]. For TMN protocol, we found two new multi-session attacks [24]. In fact, our new attacks in TMN protocol are quite surprisingly since TMN have been analyzed quite extensively [9, 11, 13, 18, 27-28]. Our preliminary results were reported in [21-24], but this paper extends our previous works by not only generalizing them into the two new techniques, which are the *on-the-fly* trace generation and the *textual* trace analysis, but also analyzing the comparative performance between the two trace generation methods.

In section 2, we provide the background on Micali's ECS1 and TMN protocol. In section 3, we compare our new method with existing related works. In section 4, we present our new CPN methodology and apply it to the two case studies.

2 Background

We use the following notations throughout the paper. $S \rightarrow R : M$ means that user S sends message M to user R. $SIG_X(M)$ represents party X's signature on a message M and we assume that M is always retrievable from $SIG_X(M)$. The encryption of a message M with party X's public key is denoted by $ENC_X(M)$. Also, H(C) stands for the hash of message C, and $E_K(M)$ means symmetric encryption on message M by key K. Note that a single session means the single execution of the protocol whereas multi-sessions mean the multiple and concurrent executions of the protocol.

2.1 TMN Authenticated Key Exchange Protocol (TMN) [18]

TMN is a cryptographic key exchange protocol for mobile communication system. TMN allows user A to exchange a session key with user B by the help of server J. The user A is called an initiator, but the user B is called a responder. The detail of TMN is described as follows.

1. $A \rightarrow J : (B, ENC_J(K_{aj})), A$
2. $J \rightarrow B : A$
3. $B \rightarrow J : (A, ENC_J(K_{ab})), B$
4. $J \rightarrow A : B, E_{K_{aj}}(K_{ab})$

Where K_{ab} is an exchanged session key and K_{aj} is A's secret which is used to transport the session key at the last step. Note that the session key is created by user B. In [18], it is suggested that the one-time pad and RSA algorithm are used as the underlying symmetric encryption and the public key encryption, respectively.

2.2 Micali's Contract Signing Protocol (ECS1) [17]

Micali proposed an efficient optimistic fair exchange protocol for contract signing. The protocol aims to ensure that two exchanging parties get each other commitment

on an agreed contract or neither of them does. There are three kinds of parties in the protocol : Alice as an initiator of the protocol, Bob as an responder of the protocol and a third trusted party who resolves a dispute between Alice and Bob during the exchange.

We denote Alice, Bob and a trusted party by A, B and TTP, respectively. It is assumed that both Alice and Bob have already agreed on a plaintext contract C before the exchange. Alice is committed to contract C as an initiator if Bob has both $SIG_A(C,Z)$ and M where $Z=ENC_{TTP}(A,B,M)$ and M is random. On the other hand, Bob is committed to C as a responder if Alice has both $SIG_B(C,Z)$ and $SIG_B(Z)$. However, there is no need for Alice to verify Z to prove Bob's commitment.

The following is the detail of a slightly modified version [25] of the original protocol to strengthen the dispute resolution request at step (4).

A1: 1) A→B: $SIG_A(C,Z)$
B1: 2) B→A: $SIG_B(C,Z)$, $SIG_B(Z)$
A2: If Bob's signatures in step 2 are both valid, then
   3) A→B: M
B2: If Bob receives valid M such that $Z=ENC_{TTP}(A,B,M)$
   then the exchange is completed
   else Bob requests TTP to resolve a dispute by the following step
      4) B→TTP: $SIG_A(C,Z)$, $SIG_B(C,Z)$, $SIG_B(Z)$
To resolve the dispute, TTP performs the following.
TTP1: If both Alice's and Bob's signatures in step 4 are valid and $Z=ENC_{TTP}(A,B,M)$ then
   5a) TTP→A: $SIG_B(C,Z)$, $SIG_B(Z)$
   5b) TTP→B: M

3 Related works

3.1 Model Checking for Cryptographic Protocols

Many model checking methods [6-16] have been developed and applied to analyze cryptographic protocols. All of them except for NRL [15] and Proverif [16] can analyze only one attack trace of a found attack. In fact, all of them are based on the *off-the-fly* trace generation which means that an attack trace is computed after a state space is generated either partially or fully. The *off-the-fly* trace generation for all attack traces involves the searching for all paths between two states which is extremely time-consuming. Indeed, the searching for all paths can be seen as a core part of algorithms for solving the traveling salesman problem which is known to be NP-complete.

Avispa [6,7] is a research project which develops four state-of-the-art model checking methods to verify cryptographic protocols. They provide high performance analysis of protocols and a large number of protocols have been analyzed. Surprisingly, they found some new attacks in some protocols. In [8], Spin which is a widely used model checker tool is employed to analyze a cryptographic protocol. A known attack to a protocol can be detected. FDR which is a model checker for CSP has been applied to analyze many cryptographic protocols in [9,10]. It can detect many new

attacks successfully in many protocols. In [11,12], Murphi which is a general model checker has also been applied to cryptographic protocols, and it discovered new attacks in some protocols. In [13-14], Petri nets-based model checking methods are employed to analyze cryptographic protocols, and they can detect known attacks only. All of the model checking methods discussed so far can analyze only one attack trace of a found attack.

There are two model checking methods which analyze multiple attack traces of a found attack, namely NRL [15] and Proverif [16]. While NRL computes all attack traces of a found attack, Proverif explores on a restricted set of attack traces which often contains only one trace. In fact, NRL is quite inefficient partly due to the computation of all paths in the *off-the-fly* approach.

3.2 Analysis of TMN and ECS1

In [25], Bao et al. analyzed ECS1 manually and found three message replay attacks in ECS1, and one attack in a simple modification of ECS1. They also proposed an improved ECS1 which can prevent all found attacks. In [26], Zhang and Liu applied a manual model checking technique to analyze ECS1. They found one new single-session attack in Micali's ECS1 and two new multi-session attacks in Bao et. al.'s modified version of ECS1. In fact, their attacks are also independently discovered by our method, but we found more attacks. In particular, we found one new single-session attack of Micali's ECS1, two new multi-session attacks in Micali's ECS1 and three new attacks of Bao's modified version of ECS1 all of which cannot be detected by Zhang and Liu's method. Since Bao et. al.'s and Zhang and Liu's methods are done by hands, their analysis does not cover attacks thoroughly.

TMN has been analyzed quite comprehensively. In [18], Simmon analyzed TMN manually and found a multi-session attack by using the homomorphic property of the underlying public key cryptographic algorithm. In [27], three formal method approaches, namely NRL, Interrogator and Inatest, for cryptographic protocols have been applied to TMN. Both NRL and Interrogator detect a single-session attack. However, Inatest can only reproduce Simmon's attack. In [11], Murφ can reproduce Simmon's attack, and detect a new multiple session attack. In [9], CSP/FDR is used by Lowe and Roscoe to discover one new single session attack and one new multi-session attack. In [13], Al-Azzoni et. al. applied CPN to detect a variant form of the attack found by Murφ [11]. In [26], by using a manual model checking, Zhang and Liu found some variant forms of Lowe and Roscoe's attacks [9] in both a single session and multiple sessions. Even though there have been many analyses on TMN, we found two new attacks on it.

4 Our Model

4.1 Our New Methodology

In general, our model checking methodology for the analysis of cryptographic protocols consists of five steps which are (1) protocol and attacker representation, (2) state space and trace generation, (3) characterization and search for attack states, (4) attack trace extraction and (5) attack trace classification. However, our new method for

computing all attack traces of a found attack contains two novel techniques which are *on-the-fly* trace generation and *textual* trace analysis. While the *on-the-fly* trace generation is employed in steps 2 and 4, the *textual* trace analysis is used in step 5. We focus our discussion on the two techniques but also explain some relevant steps if necessary. We discuss the *on-the-fly* trace generation first.

Assuming that a protocol and an attacker model are represented. The representation depends on a model checker approach. Then, a state space is generated from the representation. During the state space generation, when a state is generated, an attack trace to the state is computed at the same time and the computed trace is stored at the state. This computation is the core of the *on-the-fly* trace generation. It is important to notice that an attack trace of a state is stored at the state itself. Conceptually, an attack trace for a state is constructed by simply extending an attack trace stored in the previous state. Thus, there is no need to always compute an attack trace from the initial state, and such computation is vey expensive.

For simplicity, we assume that each state stores only one attack trace. Thus, our state space in general may contain more number of states than the state space in the *off-the-fly* trace generation. A state which can be reached by two different attack traces in the *off-the-fly* method becomes two different states in our method. To reduce the size of a computed state space in our method, we employ a decomposition technique. In particular, we define a configuration to compute a decomposed state space. In this paper, we consider the analysis of multi-sessions of protocol execution. A configuration consists of the information for the protocol execution in a multi-session setting, for example, the identities of initiator and responder, the role of attackers, secrets and nounces in each concurrent session, and a schedule of the execution of the multiple concurrent sessions. The schedule specifies that the decomposed state space is computed for one alternating execution of multiple concurrent sessions of protocol runs only, instead of all possible alternating executions. Exploring all possible alternating executions within a state space is expensive and causes a huge state space. However, we can explore each attack scenario, eg. a specific alternating execution or a specific initiator and responder, one by one by computing a decomposed state space with a specific configuration.

After the state space is obtained, attack states for each kind of attacks are searched in the state space. An attack is characterized by a vulnerability event which is an event potentially leading to a compromise of protocols. Vulnerability events are protocol-dependent. There can be many attack states which belong to the same vulnerability event and thus the same attack. When an attack state is found in the state space, an attack trace is extracted from the state immediately. By searching for all attack states of the same attack, all attack traces of the attack can be obtained without any path searching. In other words, the computation for all attack traces is reduced to the searching for attack states which can be done efficiently. This *on-the-fly* trace generation technique provides a big improvement in the computation time of all attack traces when the number of attack traces and the number of states are large.

The number of attack traces obtained can be quite large. For example, we found 1,020 traces for an attack in the TMN protocol. So, we propose *textual* trace analysis technique to classify such large number of attack traces. Those attack traces are classified by using attack patterns. Attack patterns are minimal but necessary protocol traces for an attack. The development of an attack pattern is manual because an attack

pattern is protocol-dependent. In an attack pattern, some parts of the protocol messages are fixed due to the protocol specification, but others can be varied in some ways.

While the development of an attack pattern is manual, the attack classification is automatic. Attack traces that contain the same attack pattern are classified into the same group of attack traces. As a result, a large amount of attack traces is reduced to a reasonable amount of attack patterns which are easier to analyze. Moreover, the attack classification process is iterative in that when a new attack pattern is found, it is used together with the existing patterns to filter the remaining attack traces. By using our two new techniques, protocol designers obtain a deep and thorough analysis of all possible attacks to cryptographic protocols.

Our two new techniques are independent to model checking tools. We implement them in a model checker tool called CPNTools [19,20]. Originally, CPNTools provides the *off-the-fly* trace generation only and the search mechanism for only one attack trace. We employ the simplest approach to implement the *on-the-fly* trace generation in CPNTools by using a protocol representation which records incrementally a protocol trace by users and attackers into each state. The *textual* trace analysis for attack classification is realized in CPNTools by writing an ML-like program to extract attack traces from attack states and process them.

4.2 Our Analysis for TMN

Our method for TMN. In this section, we discuss the assumptions of our protocol analysis. We also describe vulnerability events of TMN, and provide a definition of a configuration of the protocol execution. Finally, we discuss attack patterns.

Definition 1: The assumptions of the protocol execution
The following are the assumptions of the execution of the TMN protocol.

1. There are three users who are an initiator, a responder and a server. And all the users follow the protocol specification strictly and honestly.
2. There is one attacker whose abilities are defined below.
3. The underlying encryption is perfect in that nothing can be inferred from a ciphertext without the knowledge of the correct key. This is known as Dolev and Yao's assumption [29]. Also, we consider a general public key encryption scheme, instead of RSA algorithm.
4. We consider the execution of two concurrent sessions of the protocol where such execution can be performed in an alternating and non-sequential style.
5. Initiator and responder involve in one session only, but the server may involve in more than one session.
6. In a session, there must be at least one authentic user.

In assumption 5), the sessions that initiator and responder involve may not be the same. In 6), it means that there is at least one victim user in a session.

Definition 2: The attacker abilities
The attacker in our model is capable of the following:

1. The attacker can eavesdrop, modify and drop messages during the transmission between users.
2. The attacker can send any message to a user.
3. The attacker can either initiate a new session with users or take part in an existing session with users.
4. The attacker can impersonate any user.
5. The attacker can perform any cryptographic computation, eg. encryption and decryption, by using known keys, known messages and known ciphertexts with a reasonable power.
6. The attacker does not attack himself.
7. There is at most one attacker who performs the attack ability in 1) on a protocol step in a session.

The assumptions 1) and 4) mean that the attacker can act as an external observer or an impersonator, respectively. In 7), any message that is sent from an attacker will not be modified further by any other attacker.

Attack states are characterized by vulnerability events. For the TMN protocol, there are two basic vulnerability events which are secret disclosure by an attacker and session key commitment by initiator and responder. Based on the two basic events, the following combined and interesting vulnerability events can be created.

Definition 3: The combined and interesting vulnerability events.
There are three combined vulnerability events.

1. The attacker learns K_{ab} and K_{aj}, and both A and B commit on K_{ab}. $[K_{ab}, K_{aj}][K_{ab}][K_{ab}]$
2. The attacker learns K_{ab} and K_{aj}, and A is fooled to commit on K_i but B commits on K_{ab}. $[K_{ab}, K_{aj}][K_i][K_{ab}]$
3. The attacker learns K_{ab} and K_{aj}, and A is fooled to commit on K_{aj} but B commits on K_{ab}. $[K_{ab}, K_{aj}][K_{aj}][K_{ab}]$

We use the notation $[KB_1][KB_2][KB_3]$ to describe each combined vulnerability event where KB_1 stands for keys that are known by the attacker, and KB_2 and KB_3 stands for keys that are committed by users A and B, respectively, at the completion of the protocol.

In the combined event 1, the attacker learns all later communication between A and B, because the attacker obtains the session key between A and B. Lowe and Roscoe's multi-session attack [9] belongs to this event. The combined events 2 and 3 are our new attacks. The result of the events 2 and 3 can be seen as a kind of the *man-in-the-middle* attacks where the attacker situates between A and B. In event 2, the attacker can impersonate B to A by using key K_i, while the attacker can impersonate A to B by using key K_{ab}. Then, the attacker learns the later communication between A and B. The event 3 is similar to the event 2, but B impersonation to A is done by using key K_{aj}.

In the following, we provide the definition of a configuration for computing a decomposed state space for TMN protocol.

Definition 4: A configuration of the state space computation for TMN
A configuration of a decomposed state space computation consists of *((S_1, S_2,…,S_n), Sch, Tr)* and $S_i = (s,I,R,T,K,N)$ for $1 \leq i \leq n$ where n is the number of sessions, and

1. S_i is a session information for the *i*-th session which consists of
   1.1. *s* is a session identity
   1.2. *I, R* and *T* are identities for an initiator, a responder and a server, respectively.
   1.3. *K* is keys for each party (including attacker) which consists of a pair of public and private keys, and a shared key with a specific party
   1.4. *N* is nounces used by each party
2. *Sch* is a multi-session schedule which contains a specific alternating execution of multiple concurrent sessions of protocol runs
3. *Tr* is a list of attack traces and their vulnerability events

In the configuration, S_i and *Sch* are input parameters for the state space computation while *Tr* is the output from the state space computation. In this paper, we consider the multi-session schedule for the *man-in-the-middle* attack [30] where the attacker participates in two sessions and replays messages between them synchronously.

We consider the following four configurations of two concurrent sessions which are all possible configurations regarding to our assumptions. Note that in the configurations, *K*, *N*, *Sch* and *Tr* are omitted for simplicity.

1. *(1,A,B,J) & (2,In,In,J)*
2. *(1,A,In,J) & (2,In,B,J)*
3. *(1,In,B,J) & (2,A,In,J)*
4. *(1,In,In,J) & (2,A,B,J)*

There are two roles of our attacker which are an external observer and an impersonator. In configuration *(1,A,B,J)*, the attacker behaves explicitly as an external observer on the communication amongst A, B and J, In *(1,A,In,J)* and *(1,In,B,J),* the attacker explicitly impersonates B and A, respectively. But in any configuration, the attacker can impersonate implicitly any users according to our attacker model.

In TMN, an attack pattern for a session consists of the four protocol steps where ciphertexts in steps 1, 3 and 4 are of appropriate types of encryption and they are obtained from any plaintexts. But identities of initiator in step 2 and responder in step 4 are fixed to A and B, respectively. Also, identities of initiator and responder between steps 1 and 3 must be consistent, but can be anything. In fact, the important parts of the attack pattern for TMN are ciphertexts in steps 1, 3 and 4 since they contains session and encryption keys that attackers want to disclose and to forge to compromise the system, respectively. Thus, we have an attack pattern for each possible plaintext of the ciphertexts in the three steps. The following shows one attack pattern of our new attack which corresponds to the combined event 2.

1) $A \rightarrow In(J) : (B, \{K_{aj}\}PK\text{-}J), A$
   $In(J) \rightarrow J : (X2, \{K_i\}PK\text{-}J), X1$
1') $In(A) \rightarrow J : (X4, \{K_i\}PK\text{-}J), X3$

2') $J \rightarrow In(B) : X3$
2) $J \rightarrow In(B) : X1$
$In(B) \rightarrow B : A$
3) $B \rightarrow J : (X1, \{K_{ab}\}PK\text{-}J), X2$
3') $In(B) \rightarrow J : (X3, \{K_{aj}\}PK\text{-}J), X4$
4') $J \rightarrow In(A) : X4, E_{Ki}(K_{aj})$
4) $J \rightarrow In(A) : X2, E_{Ki}(K_{ab})$
$In(A) \rightarrow A : B, E_{Kaj}(K_i)$

where K_i is attacker's secret keys.

While 1) – 4) describe protocol steps in the 1st session, 1') – 4') indicate protocol steps in the 2nd session. *X1, X2, X3* and *X4* stand for arbitrary identities that the attacker creates. In step 1), the message that A sends to J is modified by the attacker. The original message is indicated by $A \rightarrow In(J)$, but the modified message by the attacker is indicated by $In(J) \rightarrow J$. Also, the messages at steps 2) and 4) are modified by the attacker.

We found 10 attack patterns for each of the events 2 and 3 which are our new attacks. The details of the attack patterns can be found in [24].

Performance. In the following, we compare the results between our *on-the-fly* and the *off-the-fly* trace generation methods both of which are implemented in CPNTools model checker. The experiment is done by using a PC with Intel Core2 Duo 2.33 Ghz and 2 GB of RAM.

In table 1, we show the comparison of the sizes of the state spaces between the two methods for the four configurations. In the configurations, session and server identities are ignored. In the worst case the number of states and arcs in the *on-the-fly* method are increased for 40.5 % and 37.9 %, respectively. However, in the best case the number of states and arcs are increased for only 9.8 % and 6.2%, respectively.

In tables 2 and 3, we compare the computation times for state spaces (*st*) and traces (*tr*) in the two methods for two configurations. Tables 2 and 3 are for the cases of the large number and the small number of states, respectively. The event 4 represented by $[K_{aj}][K_{aj}][K_{ab}]$ means that the attacker learns A's secret and fools A to commit to A's secret as a session key. The events R1 and R2 are the remaining vulnerability events where the attacker learns K_{ab} and K_{aj}, respectively. Note that in the events 2 and 3 in table 3, information is omitted since no attack trace is found for the events.

It is clear that our *on-the-fly* method improves the total computation times tremendously. When the numbers of states and traces are large, for example in the event R2 of table 2, it takes about 16 minutes (1,002 sec.) in our method, but about 19 hours (68,913 sec.) in the *off-the-fly* method. When the numbers of states and traces are small, for example in the event 4 of table 3, it takes about 2 minutes in our method, but about 11 minutes (5,376 sec.) in the *off-the-fly* method.

Indeed, our *on-the-fly* method requires more times for state space computation, but the *off-the-fly* method requires more times for trace generation. However, the time for trace generation in the *off-the-fly* exceeds greatly the time for state space computation in the *on-the-fly* method. It should be noticed that in both tables, when the number of traces is increased, the time for trace generation in the *off-the-fly* method grows greatly, but the time for trace generation in our method is almost constant.

Table 1. The comparison of the sizes of state spaces

Configurations	On-the-fly		Off-the-fly		Increment (%)	
	nodes	arcs	nodes	arcs	nodes	arcs
1. (A,B)(In,In)	104,346	109,476	74,244	79,344	40.5	37.9
2. (A,In)(In,B)	73,806	77,568	55,656	59,730	32.6	29.8
3. (In,B)(A,In)	51,212	52,639	46,637	49,543	9.8	6.2
4. (In,In)(A,B)	34,160	35,095	30,974	33,061	10.2	6.15

Table 2. The comparison of the computation times for configuration *(1,A,B,J) & (2,In,In,J)*

Events	Attack Traces	On-the-fly Time (sec)			Off-the-fly Time (sec)			Improvement %
		st	tr	total	st	tr	total	
1. Event 2	360	976	0	976	369	1839	2208	126
2. Event 3	360	976	0	976	369	1774	2143	119.56
3. Event 4	1,020	976	0	976	369	5239	5608	474
4. Event R1	8,226	976	10	986	369	40028	40397	4,039
5. Event R2	13,056	976	26	1002	369	68544	68913	6,777

Table 3. The comparison of the computation times for configuration *(1,In,In,J) & (2,A,B,J)*

Events	Attack Traces	On-the-fly Time (sec)			Off-the-fly Time(sec)			Improvement %
		st	tr	Total	st	tr	total	
1. Event 2	0	-	-	-	-	-	-	-
2. Event 3	0	-	-	-	-	-	-	-
3. Event 4	360	120	0	120	80	568	688	473.33
4. Event R1	684	120	0	120	80	1556	1,636	1,263.33
5. Event R2	2,388	120	1	121	80	5296	5,376	4,380

4.3 Our Analysis for ECS1

Our method for ECS1. Our method for the analysis of ECS1 is similar to that for TMN discussed previously. So, we discuss only the main differences between them here.

The assumptions of the protocol execution for ECS1 are similar to those assumptions in definition 1 except for assumption 5. For ECS1, the same initiator and responder may participate in more than one session. We assume two kinds of attackers: *I* and *Ar*. The attacker *I* is exactly the same as the attacker *In* discussed in definition 2. However, *Ar* is different and is a malicious user who participates in a session and conspires with attacker *I* by sharing some information. More specifically, *Ar* can be either an initiator or a responder, but not an external observer. Note that one attack found by Bao et. al. [24] involves these two kinds of attackers.

There is one vulnerability event in ECS1 protocol which is an unfair exchange state. An unfair state means that one party, who is either initiator or responder, gets

another party commitment, but the latter does not get the former commitment. There are two unfair states.

- The initiator has the responder's commitment, but the responder does not have the initiator's commitment.
- The responder has the initiator's commitment, but the initiator does not have the responder's commitment.

We found one new single-session attack and two new multi-session attacks of Micali's ECS1, and three new attacks of Bao's modified version of ECS1. The details can be found in [22].

Performance. In the following, we compare the results between the two methods implemented in CPNTools. The experiment is done by using a notebook computer with Intel Core2 Duo 2 Ghz and 3 GB of RAM.

Table 4. The comparison of the sizes of state spaces

Configuration	On-the-fly		Off-the-fly		Increment %	
	nodes	Arcs	nodes	arcs	node	arc
1.(I,Ar,c1,mi1)(I,B,c1,mi2)	235564	235563	235564	235563	0	0
2.(I,B,c1,mi1)(I,Ar,c1,mi1)	118774	119049	118774	119049	0	0
3.(I,B,c1,mi1)(Ar,I,c1,mi2)	92498	92497	92498	92497	0	0
4.(Ar,I,c1,mi1)(I,B,c1,mi2)	86470	86469	85582	85705	1.03	0.89
5.(I,B,c1,mi1)(I,B,c2,mi2)	70082	70081	68509	68629	2.29	2.11
6.(I,B,c1,mi1) (Ar,I,c1,mi1)	68694	68693	68110	68173	0.85	0.76
7.(I,B,c1,mi1) (Ar,I,c2,mi1)	68694	68693	68110	68173	0.85	0.76
8.(I,B,c1,mi1)(I,B,c1,mi1)	48728	49355	38828	39488	25.5	24.9
9.(A,B,c1,ma1)(I,Ar,c2,mi1)	34930	34929	34930	34929	0	0

In table 4, we show the comparison of the sizes of the state spaces between the *on-the-fly* and the *off-the-fly* trace generations for some configurations. Each configuration consists of the information of two concurrent sessions, and each session is represented by (i_1,i_2,c,m) where i_1 and i_2 are identities for initiator and responder, *c* is a contract and *m* is the random. The table shows that in most cases the number of states and arcs in the two methods are identical. However, in the worst case the number of states and arcs are increased for only 25.5 % and 24.9%, respectively.

In table 5, we compare the computation times for state spaces (*St*) and traces (*Tr*) in the two methods for the same configurations as table 4. The attack traces in the table are for the two unfair states in the vulnerability event.

It is clear that our *on-the-fly* method improves the total computation times greatly when the number of states and traces are large. In particular, for the best case the improvement is 116.75%. However, when the number of states and traces are small in some cases, for example in the configuration 9 which contains 34,930 nodes, the *on-the-fly* method performs better. Note that when the number of attack traces is increased, the time for trace generation in the *off-the-fly* method grows greatly, but the time in our method grows very slowly.

Table 5. The comparison of the computation times

Configuration	attack traces	On-the-fly Time (sec)			Off-the-fly Time(sec)			Improve-ment %
		St	Tr	Total	St	Tr	Total	
1.(I,Ar)(I,B)	7,032	9863	307	10,170	7090	14954	22,044	116.75
2.(I,B)(I,Ar)	2,664	2871	139	3,010	1865	3096	4,961	64.91
3.(I,B)(Ar,I)	3648	1721	114	1,835	994	1563	2,557	39.34
4.(Ar,I)(I,B)	4104	1548	125	1,673	863	1565	2,428	45.12
5.(I,B)(I,B)	1272	1091	133	1,224	694	1207	1,901	55.31
6.(I,B) (Ar,I)	1,876	1077	88	1,165	693	663	1,356	16.39
7.(I,B) (Ar,I)	1,876	1077	182	1,259	642	714	1,356	7.7
8.(I,B)(I,B)	1,116	610	42	652	251	414	665	1.99
9.(A,B)(I,Ar)	1,210	740	41	781	186	256	422	-76.69

Similar to the result in the analysis of TMN, our *on-the-fly* method requires more times for state space computation, but the *off-the-fly* method requires more times for trace generation. But here there is a case which is a configuration 9 where the time for trace generation in the *off-the-fly* does not exceed the time for state space computation in our method.

5 Discussion

According to the results obtained, we argue that our *on-the-fly* method is complementary to the *off-the-fly* method, and should be used to deal with the case for a large state space and a large number of attacks traces. Similarly, our *textual* trace analysis is also complementary to visualization technique in that when the number of traces is large, it is more suitable to employ the *textual* trace analysis. However, when the number of traces is very small, visualization technique can provide some intuitive illustration of the traces.

It is true that our *on-the-fly* trace generation method requires more amount of memory than the *off-the-fly* method. In particular, each state in our method is augmented with an attack trace. Moreover, a state which can be reached by two different attack traces in the *off-the-fly* method becomes two different states in our method. However, the *off-the-fly* method also requires a large amount of memory to store and process attack traces during the path searching for all attack traces. But our method avoids the complex path-searching computation and speeds up the whole computation time.

We hope that our very simple method would be useful for other applications of model checking for the analysis of *all errors* in any system. As a future work, we aim to optimize our method for the memory requirement, and to apply our method to analyze for other cryptographic protocols.

6 Conclusion

In this paper, we propose a very simple but practical model checking methodology for the analysis of cryptographic protocols. Our methodology offers an efficient analysis

of all attack traces for each found attack, and is independent to model checking tools. It contains two novel techniques which are *on-the-fly* trace generation and *textual* trace analysis. We apply our method to two case studies. The result shows that when the numbers of states and traces are large, our method is more efficient. In some case, our method improves the computation time over the *off-the-fly* method for 6,777%. In addition, we found many new attacks in the two case studies.

Acknowledgments. The work reported here is supported financially by National Research Council of Thailand. A previous version of CPNTools model for ECS1 was developed by Panupong Sornkom. I would like to thank Kurt Jensen and his group to suggest the names of the on-the-fly trace generation and the textual trace analysis after my presentation at CPN'09 workshop. Also, I would like to thank anonymous reviewers for their comments.

References

1. Clark, J., Jacob, J.: A survey on Authentication Protocols, Research report, University of York (1997), `http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`
2. Meadows, C.: Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends. IEEE Journal on Selected Areas in Communications 21(1), 44–54 (2003)
3. Meyer, U., Wetzel, S.: A man-in-the-middle attack on UMTS. In: The 3rd ACM workshop on Wireless Security, pp. 90–97 (2004)
4. Cervesato, I., Jaggard, A.D., Scedrov, A., Tsay, J., Walstad, C.: Breaking and fixing public-key Kerberos. Information and Computation 206(2-4), 402–424 (2008)
5. Syverson, P.F.: A Taxonomy of Replay Attacks. In: The 7th IEEE Computer Security Foundations Workshop, pp. 187–191 (1994)
6. The AVISPA project, `http://avispa-project.org`
7. Viganò, L.: Automated Security Protocol Analysis With the AVISPA Tool. Electr. Notes Theor. Comput. Sci. 155, 61–86 (2006)
8. Maggi, P., Sisto, R.: Using SPIN to Verify Security Properties of Cryptographic Protocols. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 85–87. Springer, Heidelberg (2002)
9. Lowe, G., Roscoe, B.: Using CSP to Detect Errors in the TMN Protocol. IEEE Transactions on Software Engineering 23(10), 659–669 (1997)
10. Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. Software - Concepts and Tools 17(3), 93–102 (1996)
11. Mitchell, J., Mitchell, M., Stern, U.: Automated analysis of cryptographic protocols using Murφ. In: 1997 IEEE Symposium on Security and privacy, pp. 141–151 (1997)
12. Shmatikov, V., Mitchell, J.C.: Finite-state analysis of two contract signing protocols. Theor. Comput. Sci. 283(2), 419–450 (2002)
13. Al-Azzoni, I., Down, D.G., Khedri, R.: Modeling and Verification of Cryptographic Protocols Using Coloured Petri Nets and Design/CPN. Nordic Journal of Computing 12(3), 201–228 (2005)
14. Bouroulet, R., Devillers, R., Klaudel, H., Pelz, E., Pommereau, F.: Modeling and Analysis of Security Protocols Using Role Based Specifications and Petri Nets. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 72–91. Springer, Heidelberg (2008)

15. Meadows, C.: The NRL protocol analyzer: An overview. Journal of Logic Programming 26(2), 113–131 (1996)
16. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE Computer Security Foundations Workshop, pp. 82–96. IEEE press, Los Alamitos (2001)
17. Micali, S.: Simple and Fast Optimistic Protocols for Fair Electronics Exchange. In: 21st Symposium on Principles of Distributed Computing, pp. 12–19. ACM Press, New York (2003)
18. Tatebayashi, M., Matsuzaki, N., Newman, D.: Key Distribution Protocol for Digital Mobile Communication Systems. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 324–334. Springer, Heidelberg (1990)
19. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Monographs in Theoretical Computer Science, vol. 1. Springer, Heidelberg (1997)
20. CPNTools, `http://wiki.daimi.au.dk/cpntools/`
21. Sornkhom, P., Permpoontanalarp, Y.: Security Analysis of Micali's Fair Contract Signing Protocol by Using Coloured Petri Nets. In: The 9th ACIS-SNPD 2008, pp. 329–334. IEEE Press, Thailand (2008)
22. Sornkhom, P., Permpoontanalarp, Y.: Security Analysis of Micali's Fair Contract Signing Protocol by Using Coloured Petri Nets: Multi-session case. In: The 5th International Workshop on Security in Systems and Networks, pp. 1–8. IEEE press, Italy (2009)
23. Permpoontanalarp, Y., Sornkhom, P.: A New Coloured Petri Net Methodology for the Security Analysis of Cryptographic Protocols. In: The 10th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Denmark, pp. 81–100 (2009)
24. Permpoontanalarp, Y.: Security Analysis of the TMN protocol by using Coloured Petri Nets: Multi-Session Case. In: The 10th International Conference on Intelligent Technologies, China, pp. 401–410 (2009)
25. Bao, F., Wang, G., Zhou, J., Zhu, Z.: Analysis and Improvement of Micali's Fair Contract Signing Protocol. In: Wang, H., Pieprzyk, J., Varadharajan, V. (eds.) ACISP 2004. LNCS, vol. 3108, pp. 176–187. Springer, Heidelberg (2004)
26. Zhang, Y., Wang, Z., Yang, B.: The Running-Mode Analysis of Two-Party Optimistic Fair Exchange Protocols. In: Hao, Y., Liu, J., Wang, Y.-P., Cheung, Y.-m., Yin, H., Jiao, L., Ma, J., Jiao, Y.-C. (eds.) CIS 2005. LNCS (LNAI), vol. 3802, pp. 137–142. Springer, Heidelberg (2005)
27. Kemmerer, R., Meadows, C., Millen, J.: Three systems for cryptographic protocol analysis. Journal of Cryptology 7(2) (1994)
28. Zhang, Y., Liu, X.: An approach to the formal analysis of TMN protocol. In: Chen, K. (ed.) Progress on Cryptography: 25 years of Cryptography in China. The International Series in Engineering and Computer Science, vol. 769, pp. 235–243. Springer, Netherlands (2004)
29. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–207 (1983)
30. Lowe, G.: An Attack on the Needham-Schroeder Public-Key Authentication Protocol. Information Processing Letters 56(3), 131–133 (1995)
31. ITU-T, Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva (1996)

On Efficient Models for Model Checking Message-Passing Distributed Protocols*

Péter Bokor, Marco Serafini, and Neeraj Suri

Technische Universität Darmstadt, Germany
{pbokor,marco,suri}@cs.tu-darmstadt.de

Abstract. The complexity of distributed algorithms, such as state machine replication, motivates the use of formal methods to assist correctness verification. The design of the formal model of an algorithm directly affects the efficiency of the analysis. Therefore, it is desirable that this model does not add "unnecessary" complexity to the analysis. In this paper, we consider a general message-passing (MP) model of distributed algorithms and compare different ways of modeling the message traffic. We prove that the different MP models are *equivalent* with respect to the common properties of distributed algorithms. Therefore, one can select the model which is best suited for the applied verification technique.

We consider MP models which differ regarding whether (1) the event of message delivery can be interleaved with other events and (2) a computation event must consume all messages that have been delivered after the last computation event of the same process. For generalized MP distributed protocols and especially focusing on fault-tolerance, we show that our proposed model (without interleaved delivery events and with relaxed semantics of computation events) is significantly more efficient for explicit state model checking. For example, the model size of the Paxos algorithm is $1/13^{th}$ that of existing equivalent MP models.

1 Introduction

The use of distributed, message-passing (MP) protocols is an increasingly advocated approach for performance and availability objectives across the spectrum of service and safety critical systems, e.g., Google File System [5] or Microsoft's state machine replication [14]. Due to the complexity of MP protocols, automated tools are desired for the debugging and verification of these protocols. *Model checking* (MC) [8] is suggested as a good formal analysis candidate given its ability to prove complex properties and especially to find bugs. However, MC is often restricted by state space explosion, i.e., when detailed models require a prohibitively large number of states to explore. Therefore, we expect from the formal model that it does not introduce unnecessary complexity and, at the same time, provides a faithful representation of the system.

We start from an established model of MP algorithms [1], written as **MP**, where two kinds of events are defined: *computation events* that receive/send messages and update the local state of the executing process, and *delivery events* that move messages from output to input buffers representing message channels. Typically, an MP algorithm is supposed to implement an *abstraction*, e.g., of reliable communication, an accessible

* Research supported in part by Microsoft Research, IBM Faculty Award.

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 216–223, 2010.
© IFIP International Federation for Information Processing 2010

and correct server or register, and the specification of this abstraction is independent of the underlying MP system. In other words, delivery events are "invisible" with respect to the desired properties of common MP algorithms. In this paper, we propose alternative semantics of delivery events and show that they result in equivalent models if delivery events are invisible. One of the proposed semantics turns out to be particularly efficient for MC as it yields smaller models than other semantics, as shown in our experiments.

The original model **MP** allows events to be interleaved arbitrarily. For example, consider the following run which consists of two computation and one delivery events: "process p_i sends message m_i to p_j"; "p_k sends m_k to p_j"; "m_i is delivered". Note that the delivery event corresponding to m_i is interleaved with the sending of m_k. **MP** defines that computation events must empty all local input buffers (restricted computation semantic). For example, any computation event at process p_j extending the previous run must consume m_i. In an attempt to find an abstraction which faithfully and effectively models MP algorithms, we deal with the following two questions.

*First, is **MP** a general model, i.e., does the restricted computation semantic limits to a class of systems?* In fact, in our first proposed model, termed as **M-MP**, we relax the computation semantic of **MP** and allow computation events to empty a *subset* of all local input buffers. As a result, a computation event at process p_j extending the run above can also be an internal event and needs not necessarily process m_i. In this way, **M-MP** adds a new source of non-determinism compared to **MP**. However, we prove that **MP** and **M-MP** are equivalent. Therefore, **MP** is indeed a general model.

*Second, can we obtain a model which is equivalent with **MP** but yields smaller state spaces?* It is intuitive that the interleaving semantic of delivery events results in a large number of states. Therefore, our next proposed model, called **M**, eliminates explicit delivery events such that messages that are sent by some computation event *comp* are placed in the target input buffers in an atomic step together with *comp*. Therefore, our sample run is not a valid run in **M**. A similar, and valid, run would look like this: "p_i sends m_i to p_j and m_i is delivered"; "p_k sends m_k to p_j and m_k is delivered". Interestingly, this model together with the relaxed computation semantic result in a model equivalent with **MP** (and thus with **M-MP** too). Intuitively, this is because the non-determinism of when delivery events are scheduled is replaced by the non-determinism of deciding the set of those locally delivered messages that are processed by a computation event.

The basis of our equivalence is *stuttering* [12], a property usually used in an optimization of temporal logic MC called partial-order reduction (POR) [8]. Intuitively, two runs are stuttering equivalent if they can be partitioned into blocks where the ith block consists of subsequent system states exhibiting the same assertions in both runs. For example, if a, b, c are assertions labeling states of the system, then the runs $abc...$ and $abbc...$ are stuttering equivalent. Intuitively, we show that each block, e.g., the block of b's, corresponds to one visible computation event and the length of the block is determined by (invisible) delivery events executed in the MP model in use.

Related Work. In POR, certain runs of the system are not explored such that POR guarantees that each of these runs is stuttering equivalent with at least one the explored ones. It is in theory possible that given a model, say **M-MP**, a stuttering equivalent

model, such as **MP** or **M**, is automatically generated by POR. However, independent of the technique used, POR always has some run-time overhead.

A recent work closely related to ours is [6] where, similarly to the reduction from **MP** to **M**, it is shown that a fine-grained model of distributed MP algorithms can be reduced to a stuttering equivalent coarse-grained model. The main difference between [6] and this paper lies in the system model. In [6], a special class of MP algorithms is characterized by (1) communication-closed rounds and (2) crash faults based on the Heard-Of model [7]. The reduction theorem shows that it suffices to model a run of the system as sequence of synchronized rounds where in each round every correct process sends and receives messages and updates its local state. Our system model is a general one and it does not assume that a run is divided into rounds such that a message sent in round i must be delivered before the end of round i otherwise the message is considered to be lost (communication-closedness), neither do we restrict to crash faults.

In our models, different events can be unrestrictedly interleaved, which corresponds to asynchronous systems and also it allows "unfair" runs where certain processes can make no steps. In order to model synchronous rounds or attain fairness [8], the restriction of our general model is necessary.

The presented general model also allows the modeling of process and communication faults [3]. For example, a Byzantine process is a regular process sending arbitrary messages or lossy channels can be modeled through auxiliary computation events deleting messages from channels. We remark that the proposed formal model cannot directly mimic channel overflows because channels are defined as (infinite) sets of messages. However, channel overflow can be modeled through lossy channels.

2 The System Model

2.1 Model of Computation in Message-Passing Systems

The system consists of n processes. Processes are interconnected via directed *channels* from a set *Chan* following an arbitrary topology. If there is a channel from process i to j, process i maintains an output buffer called $outbuf_j$ which contains the messages in transit, i.e., messages that are sent by process i to j but not yet delivered by the channel. Similarly, process j maintains an input buffer called $inbuf_i$ containing the messages sent by i to j and delivered by the channel. Formally, a buffer is a set of messages. By assumption, buffers are infinite and initially empty. In addition, every process i maintains a *local state*, initially taken from $Init_i$. An (initial) *configuration* of the system is a tuple $c = (p_1, ..., p_n)$ where p_i stores process i's (initial) local state and its input and output buffers. We write $proc(c) = (s_1, ..., s_n)$ to mean the tuple of local states of each process.

Transitions between configurations are modeled via *computation events*. The set of all computation events is denoted by $Comp$. Every computation event $comp$ is associated with a process i and a finite set M of messages. The effect of $comp$ on the current configuration is deterministic and is defined as follows: every messages M is removed from i's input buffers, i's local state s_i is updated, and at most one message is added to every output buffer of i.[1] We say that $comp$ is *enabled* in a configuration c if M

[1] Note that computation events with $M = \emptyset$ can be used to model internal events.

is a subset of the union of all input buffers of process i and the update of process i's local state is defined by $comp$. Non-determinism can be modeled through concurrently enabled computation events.

Based on the previous definitions, a *program* is a tuple $(n, \mathit{Chan}, \mathit{Comp})$. As we will now see, we have multiple choices to model the *delivery* of messages.

2.2 Message-Passing Models

*The **MP** Model [1].* This is an existing model where special events, called *delivery events*, are used to move a message from an output to the corresponding input buffer. Formally, a delivery event of a message sent by process i to j is denoted as $del(i, j, m)$ which means that message m is removed from outbuf_j of process i and placed into inbuf_i of process j. Event $del(i, j, m)$ is *enabled* if m is in outbuf_j of process i.

In addition, the **MP** model defines that every computation event $comp$ associated with some process i must empty *all* input buffers of i (restricted computation semantic). This means that if $comp$ is executed in a configuration c and M is the union of all input buffers of process i in c, then M is the set of messages that is associated with $comp$.

*The **M-MP** Model.* In an attempt to find out whether the restricted computation semantic means a real restriction, we define a new model called **M-MP** which is similar to **MP** but, given the union M of all input buffers of process i in configuration c, a computation event $comp$ associated with i and message set M' can be executed in c if $comp$ is enabled in c and $M' \subseteq M$. We will see that **M-MP** is equivalent with **MP**, thus, this relaxation is unnecessary.

*The **M** Model.* Intuitively, delivery events generate lots of intermediate states where the local states of the processes are unchanged. Therefore, in the next model, called **M**, we ban delivery events and place messages directly to input buffers: messages that are sent by some computation event $comp$ are moved to the corresponding input buffers in an atomic step together with $comp$. Formally, we define a computation event $comp$ associated with process i as in **M-MP** with the exception that every message m placed by $comp$ into output buffer $output_j$ of i is placed into input buffer $input_i$ of process j. In **M**, we use the same relaxed semantic of computation events as in **M-MP**. In fact, this is key to prove that the new model **M** is equivalent with **MP** and **M-MP**.

2.3 Semantics: State Transition System

Given a program P and a message-passing model **MP**, **M-MP**, or **M**, we define a *state transition system* (STS) MP_P, $M\text{-}MP_P$, or M_P in the usual way. An STS is a tuple (S, T, S_0, L) where S is the set of states, T is a set of events such that $\alpha : S \to S$ for each $\alpha \in T$, $S_0 \subseteq S$ is the set of initial states, and $L : S \to 2^{AP}$ is the labeling function which labels each state with a set of atomic propositions, a subset of AP.

In all STSs, S is the set of all configurations and S_0 is the set of initial configurations. In MP_P and $M\text{-}MP_P$, T is the set of all computation and delivery events. In M_P, T is the set of all computation events and it contains no delivery events. For every $\alpha \in T$ and $c, c' \in S$, $\alpha(c) = c'$ if α is enabled in c and if c' is the configuration which results in executing α in c as defined by **MP**, **M-MP**, **M**, respectively.

3 Equivalent Message-Passing Models with Invisible Delivery

3.1 The Notion of Equivalence and the Structure of Our Proof

Although the models **MP**, **M-MP** and **M** differ from each other in how they model the delivery of messages, we prove that given a program P they yield STSs that preserve exactly the same set of properties written in temporal logic. In order to show the equivalence, we assume that a property can only make assertions about process states by restricting that **(A1)** for every delivery event α and configuration $c \in S$ such that α is enabled in c, $L(c) = L(\alpha(c))$.

Property Language. We adopt temporal logic [8] as the property description language. Temporal operators (such as "future", "until", etc.) are interpreted over *runs*. Formally, a run σ is a sequence of configuration $c_0 \xrightarrow{\alpha_0} c_1 \xrightarrow{\alpha_1} c_2 ...$ where c_0 is an initial configuration and $c_{i+1} = \alpha_i(c_i)$ with $\alpha_i \in T$. In this case, we call c_i a *reachable* configuration.

We ban the "next" operator from our property language. Intuitively, this is because the length of a run to reach a configuration is different depending on the MP model. For example, the delivery of multiple messages is done atomically in **M**, whereas it corresponds to a sequence of delivery events in **MP**. It is known that the use of the "next" operator can be avoided in the specification of most concurrent systems [12]. We assume that our temporal logic is linear-time (LTL). Therefore, conditions about different branches along a run cannot be specified (in contrast to CTL). This is not a limitation because the common properties of distributed protocols specify that *every* (fair) run must satisfy the property. The logic we have just described is the well-known next-free LTL (or LTL-X) [8].

Property Preservation. The key to our equivalence results is that LTL-X cannot distinguish between *stuttering equivalent* runs [12]: given an LTL-X formula f and two stuttering equivalent runs σ and σ', f holds along σ if and only if it holds along σ' [8]. Formally, two runs $\sigma = c_0 \xrightarrow{\alpha_0} c_1 ...$ and $\sigma' = c'_0 \xrightarrow{\beta_0} c'_1 ...$ are stuttering equivalent, $\sigma \approx_{st} \sigma'$ in short, if there are two infinite sequences of integers $0 = i_0 < i_1 < ...$ and $0 = j_0 < j_1 < ...$ such that for every $k \geq 0$, $L(c_{i_k}) = L(c_{i_k+1}) = ... = L(c_{i_{k+1}-1}) = L(c'_{j_k}) = L(c'_{j_k+1}) = ... = L(c'_{j_{k+1}-1})$.

Proof Structure. In order to prove stuttering equivalence between two STSs A and B we have to show for *every* run in A a stuttering equivalent run in B and vice versa. In case of three STSs, corresponding to a program P and three MP models, this means at most six proofs. In order to minimize the number of proofs we utilize that every run according to the **MP** model is also a valid run according to **M-MP**. Therefore, if we prove that for every run according to **M** there is a stuttering equivalent run according to **MP** (Figure 1(a)) and for every run according to **M-MP** there is a stuttering equivalent run according to **M** (Figure 1(b)), then we know that the model **M** is equivalent with **MP** and **M-MP**. In addition, the transitivity of stuttering equivalence implies that the models **MP** and **M-MP** are also equivalent. In summary, our equivalence results imply the following property preservation.

Corollary 1. *Given a program P and an LTL-X formula f such that* ***A1*** *holds, let $MP_P, M\text{-}MP_P$ and M_P be the STS corresponding to P under the message-passing*

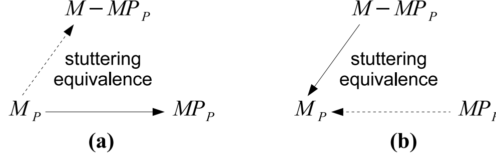


Fig. 1. Equivalence of different MP models given program P and assumption **A1**. Solid line directions are proven by (a) Theorem 1 and (b) Theorem 2 while dashed line arrows are implications thereof. The equivalence is completed by the transitivity of stuttering as stated in Theorem 3.

*model **MP**, **M-MP** and **M**, respectively. Then, f holds in MP_P iff it holds in $M\text{-}MP_P$ and iff it holds in M_P.*

In the subsequent Subsection, we precisely state and explain the main ideas of the Theorems that imply Corollary 1. The proofs of these Theorems can be found in our technical report [4].

3.2 Stuttering Equivalent Paths in MP, M-MP and M

As explained above, we have to find for each run σ_M in M_P a run σ_{MP} in MP_P such that σ_{MP} is stuttering equivalent with σ_M. Similarly, for every run $\sigma_{M\text{-}MP}$ in $M\text{-}MP_P$ we need to show a stuttering equivalent run σ_M in M_P.

First, consider $\sigma_M = c_0 \xrightarrow{\alpha_0} c_1 \xrightarrow{\alpha_1} c_2 \ldots$. Since computation events in σ_M may leave messages in the input buffers of the associated process, we construct σ_{MP} such that it performs the same sequence of computation events as σ_M but delivers messages "on demand". This means that, given two subsequent computation events α_i and α_{i+1} ($i \geq 0$), α_i and α_{i+1} occur in σ_{MP} in the same order as in σ_M and α_{i+1} is directly preceded by a sequence of exactly those delivery events that are consumed by α_{i+1}. In this way, the restricted computation semantic can be respected. We have to prove that this construction of σ_{MP} is always possible and that, as delivery events are invisible, $\sigma_M \approx_{st} \sigma_{MP}$ holds. Formally, we have the following result.

Theorem 1. *Given a program P and a run $\sigma_M = c_0 \xrightarrow{\alpha_0} c_1 \ldots$ in M_P, a run $\sigma_{MP} = c'_0 \xrightarrow{\beta_0} c'_1 \ldots$ in MP_P can be constructed as follows. Initially, $c'_0 = c_0$. Furthermore, for every $i = 0, 1, \ldots$ and α_i, execute (in arbitrary order) a delivery event β_j for each message that is consumed by α_i in σ_M, and then execute α_i in σ_{MP}. In addition, $\sigma_M \approx_{st} \sigma_{MP}$.*

Second, consider $\sigma_{M\text{-}MP} = c_0 \xrightarrow{\alpha_0} c_1 \xrightarrow{\alpha_1} c_2 \ldots$ where α_i is a delivery or a computation event. Intuitively, σ_M is the same as $\sigma_{M\text{-}MP}$ without delivery events. This means that although in σ_M all messages are delivered atomically it is possible, according to **M**'s semantics, to empty only a subset of messages delivered at each process. Similarly to the previous case, $\sigma_{M\text{-}MP}$ and σ_M are stuttering equivalent because delivery events are invisible and the sequence of computation events is identical in the two runs.

Theorem 2. *Given a program P and a run $\sigma_{M\text{-}MP} = c_0 \xrightarrow{\alpha_0} c_1...$ in $M\text{-}MP_P$, a run $\sigma_M = c'_0 \xrightarrow{\beta_0} c'_1...$ in M_P can be constructed as follows. Initially, $c'_0 = c_0$. Furthermore, $\beta_1, \beta_2, ...$ is the sequence of all computation events of $\sigma_{M\text{-}MP}$ in the order as they appear in $\sigma_{M\text{-}MP}$. In addition, $\sigma_{M\text{-}MP} \approx_{st} \sigma_M$.*

Theorems 1 and 2 directly imply the stuttering equivalence of $M\text{-}MP_P$ and MP_P if we can show the transitivity of stuttering equivalence. The proof is based on the simple observation that there might be multiple partitioning, i.e., integer sequences $i_0, i_1, ...$ and $j_0, j_1, ...$, of the same pair of stuttering equivalent runs. Then, the transitivity property can be easily shown based on the partitioning where adjacent segments never have the same labels. Formally, we have the following result.

Theorem 3. *Given an STS and three runs $\sigma, \sigma', \sigma''$, $\sigma \approx_{st} \sigma'$ and $\sigma' \approx_{st} \sigma''$ imply that $\sigma \approx_{st} \sigma''$ also holds.*

4 Evaluation

Setup. We compared the efficiency of MC with different MP models. We used the Murφ model checker [9] which supports symmetry reduction (SR) [8,13], a powerful optimization known to be very efficient for fault-tolerant (FT) distributed protocols [3]. We model checked some basic properties of the Paxos algorithm [10], a highly concurrent crash-tolerant consensus algorithm. The MC results of another FT algorithm, the Byzantine Generals [11], can be found in our technical report [4].

Paxos solves consensus, where many processes can propose a local value and only one of these values is decided. Paxos uses three symmetric roles, m leaders, n acceptors, and some learners, and assumes that at most a minority of all acceptors is crash faulty. Leaders send a proposal, composed of the current local value and a proposal number, to all acceptors. An acceptor accepts a proposal only if it has not yet received any other proposal with a higher proposal number. A proposal is termed as chosen if a majority of acceptors accepts it. A chosen proposal can be learnt by the learners by collecting the accepted proposals from the acceptors. Consensus requires that no two proposals with different values are ever chosen (safety) and that a proposal is learnt (liveness).

Results. The results of our MC experiments are shown in Table 1. These include the verification of the safety property of Paxos as well as false properties and fault-injected protocols where, for each case, a counterexample was found. Our experiments cover those (non-trivial) settings that were feasible to verify with Murφ.[2] For each case we ran six experiments with the three different MP models and without and with SR.

We observe that using **M** yields significantly fewer explored states and transitions and less time than **M-MP** and **MP**. For example, in the verification of Paxos with $n = 3$, the number of states and transitions is approximately $1/13$ of those in the STS with **M-MP**. The same factor for the verification time is $1/40$. In this example, the size of the **M**-model without SR is already smaller than other models that exploit symmetries. In addition, the verification of Paxos with $n = 4$ is only feasible with the **M**-model.

[2] All experiments ran on DETERlab machines [2] with Xeon processors and 4 GB memory.

Table 1. MC results of Paxos with Murφ by using different MP models (**M-MP**, **MP** and **M**) and symmetry reduction

Protocol	Param.	Property	Model	States	Transitions	Time	Result
Paxos	$m = 2$ $n = 3$	safety	**M-MP**	—			Out of mem.
			M-MP symm.	1,754,621	12,463,946	15 m	Verified
			MP	—			Out of mem.
			MP symm.	1,754,621	11,647,308	13 m	Verified
			M	1,577,161	11,411,586	3 m	Verified
			M symm.	135,271	980,290	24 s	Verified
		Erroneous safety (chosen = accepted)	**M-MP**	—			Out of mem.
			M-MP symm.	468,581	2,676,397	2 m	CE found
			MP	—			Out of mem.
			MP symm.	468,444	2,488,162	2 m	CE found
			M	476,575	2,435,659	31 s	CE found
			M symm.	49,290	256,761	8 s	CE found
Faulty Paxos (always accept proposals)		safety	**M-MP**	—			Out of mem.
			M-MP symm.	890,127	5,174,054	7 m	CE found
			MP	—			Out of mem.
			MP symm.	894,166	4,840,435	6 m	CE found
			M	1,026,203	5,598,379	1 m	CE found
			M symm.	99,781	553,159	15 s	CE found
Paxos	$m = 2$ $n = 4$	safety	**M-MP**	—			Out of mem.
			M-MP symm.	—			Out of mem.
			MP	—			Out of mem.
			MP symm.	—			Out of mem.
			M	—			Out of mem.
			M symm.	775,355	7,701,472	4 m	Verified

References

1. Attiya, H., Welch, J.: Distributed Computing. John Wiley and Sons, Chichester (2004)
2. Benzel, T., et al.: Design, Deployment, and Use of the Deter Testbed. In: Proc. DETER Community Workshop on Cyber Security Experimentation and Test (2007)
3. Bokor, P., Serafini, M., Suri, N., Veith, H.: Role-Based Symmetry Reduction of Fault-tolerant Distributed Protocols with Language Support. In: Proc. ICFEM, pp. 147–166 (2009)
4. http://www.deeds.informatik.tu-darmstadt.de/peter/MP.pdf
5. Chandra, T.D., et al.: Paxos Made Live: An Engineer. Persp. In: Proc. PODC, pp. 398–407 (2007)
6. Chaouch-Saad, M., Charron-Bost, V., Merz, S.: A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In: Bournez, O., Potapov, I. (eds.) RP 2009. LNCS, vol. 5797, pp. 93–106. Springer, Heidelberg (2009)
7. Charron-Bost, B., Schiper, A.: The Heard-Of Model: Computing in Distributed Systems with Benign Failures. Distr. Comp. (to Appear, 2009)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
9. Dill, D.L., et al.: Protocol Verif. as a Hardware Design Aid. In: Proc. ICCD, pp. 522–525 (1992)
10. Lamport, L.: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
11. Lamport, L., et al.: The Byz. Generals Problem. ACM TOPLAS 4(3), 382–401 (1982)
12. Lamport, L.: What good is Temporal Logic? In: Proc. Inf. Processing, pp. 657–667 (1983)
13. Miller, A., et al.: Symmetry in Temporal Logic MC. ACM Comp. Surv. 38(3), 8 (2006)
14. Yang, J., et al.: MODIST: Transp. MC of Unmodif. Distr. Sys. In: Proc. NSDI, pp. 213–228 (2009)

Logics for Contravariant Simulations*

Ignacio Fábregas, David de Frutos Escrig, and Miguel Palomino

Departamento de Sistemas Informáticos y Computación, UCM
fabregas@fdi.ucm.es, {defrutos,miguelpt}@sip.ucm.es

Abstract. Covariant-contravariant simulation and conformance simulation are two generalizations of the simple notion of simulation which aim at capturing the fact that it is not always the case that "the larger the number of behaviors, the better". Therefore, they can be considered to be more adequate to express the fact that a system is a correct implementation of some specification. We have previously shown that these two more elaborated notions fit well within the categorical framework developed to study the notion of simulation in a generic way. Now we show that their behaviors have also simple and natural logical characterizations, though more elaborated than those for the plain simulation semantics.

1 Introduction and Some Related Work

Simulations are a very natural way to compare systems modeled by labeled transition systems or other related mechanisms based on describing the behavior of states by means of the actions they can execute [12]. They aim at comparing processes based on the simple premise "you are better if you can do as much as me, and perhaps some additional new things". This assumes that all the executable actions are controlled by the user (hence, no difference between input and output actions) and does not take into account that the system will choose in an unpredictable internal way whenever it has several possibilities for the execution of an action; thus, the more possibilities, the less control.

In order to cope with this situation one should consider adequate versions of simulation where the meaning of actions and the idea of preferring processes that are less non-deterministic are taken into account. This leads to two new notions of simulation: covariant-contravariant simulation and conformance simulation, that we roughly sketched in [6] and presented in detail in [7], where we proved that they can be obtained as particular instances of the general notion of categorical simulation developed by Hughes and Jacobs [9].

The first new notion is that of covariant-contravariant simulation, where the alphabet of actions Act is partitioned into three disjoint sets Act^l, Act^r, and Act^{bi}. The intention is that simulations will treat the actions in Act^l like in the ordinary case, they will interchange the roles of the related processes for those

* Research supported by the Spanish projects DESAFIOS10 TIN2009-14599-C03-01, TESIS TIN2009-14321-C02-01 and PROMETIDOS S2009/TIC-1465.

J. Hatcliff and E. Zucca (Eds.): FMOODS/FORTE 2010, LNCS 6117, pp. 224–231, 2010.
© IFIP International Federation for Information Processing 2010

actions in Act^r, and they will impose a symmetric condition (like that defining bisimulation) for the actions in Act^{bi}. The second notion, conformance simulation, captures the conformance relations [10] that several authors introduced in order to formalize the notion of possible implementations.

After showing in [7] that they can be formalized as categorical simulations, in this paper we present their logical characterizations. We expect that they will contribute to clarify the meaning of the corresponding simulations, shedding light on the properties that can be established when using these two frameworks within a specification procedure.

Certainly, the distinction between input and output actions or similar classifications is not meant to be new at all and, for instance, it was present in modal transition systems as early as the end of the eighties. It also plays a central role in I/O-automata [11] and, more recently, appears as component of several works on interface automata [4], where the covariant-contravariant distinction is found when the guarantees of the specification can only be assumed if the conditions of the specification are satisfied.

Concerning conformance simulation, the first related references are also quite old [10] and correspond to the notion of conformance testing, which is close to failure semantics [13]. However, it is a bit surprising that in both cases there is lack of a basic theory where these notions are presented in a simplified scenario, stressing their main characteristics and properties.

Let us conclude this introduction by remarking that there is a large collection of recent papers where notions close to those studied here are either developed or applied. We regret not having the time or space to discuss, or even to cite, many of them and just to give a hint we point out [1,2], where several references to other preliminary works in those directions can be found.

2 Recalling Contravariant Simulations

We consider labeled transition systems (LTS) $(P, A, \rightarrow_P)$, where $\rightarrow_P \subseteq P \times A \times P$, to define the operational semantics of a family of processes $p \in P$. We say that the LTS is *finitary* when for each $p \in P$ and $a \in A$ we have $|\{p' \mid p \xrightarrow{a} p'\}| < \infty$.

We refer to [7] for a more extensive motivation of covariant-contravariant simulations; here we only comment on the case of input/output automata. To define an adequate simulation notion for them we observe that the classic approach to simulations is based on the definition of semantics for reactive systems, where all the actions of the processes correspond to input actions that the user must trigger. Instead, the situation is the opposite whenever we have explicit output actions: it is the system that produces the actions and the user who is forced to accept the produced output. Then, it is natural to conclude that in the simulation framework we have to dualize the simulation condition when considering output actions, and this is exactly what our anti-simulation relations do.

Definition 1. *Given $P = (P, A, \rightarrow_P)$ and $Q = (Q, A, \rightarrow_Q)$, two labeled transition systems for the alphabet A, and $\{A^r, A^l, A^{bi}\}$ a partition of this alphabet,*

a (A^r, A^l)***-simulation*** *(or just a covariant-contravariant simulation) between them is a relation* $S \subseteq P \times Q$ *such that for every* pSq *we have:*

- *for all* $a \in A^r \cup A^{bi}$ *and all* $p \xrightarrow{a} p'$ *there exists* $q \xrightarrow{a} q'$ *with* $p'Sq'$*.*
- *for all* $a \in A^l \cup A^{bi}$*, and all* $q \xrightarrow{a} q'$ *there exists* $p \xrightarrow{a} p'$ *with* $p'Sq'$*.*

We will write $p \lesssim_{CC} q$ *if there exists a covariant-contravariant simulation* S *such that* pSq*.*

Conformance simulations allow the extension of the set of actions offered by a process, so that in particular $a \lesssim a+b$, but they also consider that a process can be "improved" by reducing the nondeterminism in it, so that $ap+aq \lesssim ap$. In this way we have again a kind of covariant-contravariant simulation, not driven by the alphabet of actions executed by the processes but by their nondeterminism.

Definition 2. *Given* $P = (P, A, \rightarrow_P)$ *and* $Q = (Q, A, \rightarrow_Q)$ *two labeled transition systems for the alphabet* A*, a* ***conformance simulation*** *between them is a relation* $R \subseteq P \times Q$ *such that whenever* pRq*, then:*

- *For all* $a \in A$*, if* $p \xrightarrow{a}$*, then* $q \xrightarrow{a}$ *(this means, using the usual notation for process algebras, that* $I(p) \subseteq I(q)$*).*
- *For all* $a \in A$ *such that* $q \xrightarrow{a} q'$ *and* $p \xrightarrow{a}$*, there exists some* p' *with* $p \xrightarrow{a} p'$ *and* $p'Rq'$*.*

We will write $p \lesssim_{CS} q$ *if there exists a conformance simulation* R *such that* pRq*.*

3 Logical Characterizations of the New Semantics

3.1 Covariant-Contravariant Simulations

The class $\mathcal{L}_S$ characterizing the simulation semantics is defined in [3] as that containing tt, conjunctions $\bigwedge_{i \in I} \varphi_i$ (which can be just finite or binary if we only want to characterize finitary process) and the existential operator $\langle a \rangle \varphi$, whose semantics is defined by: $p \models \langle a \rangle \varphi$ if there exists some p' such that $p \xrightarrow{a} p'$ and $p' \models \varphi$.

If we compare it with the Hennessy-Milner logic $\mathcal{L}_{HM}$ [8], it can be noted that the main diference is that negation is not present. Obviously, this must be the case to capture a strict order that is not an equivalence relation, such as $\lesssim_{CC}$. However, adding both the constant ff and the disjunction $\bigvee_{i \in I} \varphi_i$ does no harm, thus obtaining $\bar{\mathcal{L}}_S$ which also characterizes $\lesssim_S$. Indeed, ff is just $\bigvee_{\emptyset} \varphi_i$, while disjunctions can be moved to the top of the expression because $\langle a \rangle \bigvee_{i \in I} \varphi_i \equiv \bigvee_{i \in I} \langle a \rangle \varphi_i$, and $p \models \bigvee_{i \in I} \varphi_i$ iff there exists some $i \in I$ such that $p \models \varphi_i$.

The inspiration to obtain the logic characterizing $\lesssim_{CC}$ comes from the fact that if we only have contravariant actions, then $\lesssim_{CC}$ becomes $\lesssim_S^{-1}$, and therefore by negating all the formulas in $\bar{\mathcal{L}}_S$ we would obtain the desired characterization. In particular, for the modal operator $\langle a \rangle$ we would obtain its dual form $[a]$, whose semantics is defined by: $p \models [a]\varphi$ *if* $p' \models \varphi$ *for all* p' *such that* $p \xrightarrow{a} p'$*.*

Then, in the presence of both covariant and contravariant actions, we need to consider the existential operator $\langle a \rangle$ for $a \in A^r \cup A^{bi}$ and the universal operator $[a]$ for $a \in A^l \cup A^{bi}$, thus obtaining the following definition.

Definition 3. *Given an alphabet A, and $\{A^r, A^l, A^{bi}\}$ a partition of this alphabet, the class $\mathcal{L}_{CC}$ of covariant-contravariant simulation formulas over A is defined recursively by:*

- tt *and* ff *are in* $\mathcal{L}_{CC}$.
- *If I is a set and $\varphi_i \in \mathcal{L}_{CC}$ for all $i \in I$ then $\bigwedge_{i\in I} \varphi_i \in \mathcal{L}_{CC}$, $\bigvee_{i\in I} \varphi_i \in \mathcal{L}_{CC}$.*
- *If $\varphi \in \mathcal{L}_{CC}$ and $a \in A^r \cup A^{bi}$ then $\langle a \rangle\varphi \in \mathcal{L}_{CC}$.*
- *If $\varphi \in \mathcal{L}_{CC}$ and $a \in A^l \cup A^{bi}$ then $[a]\varphi \in \mathcal{L}_{CC}$.*

The satisfaction relation $\models$ is defined recursively by:

- $p \models \mathsf{tt}$.
- $p \models \bigwedge_{i\in I} \varphi_i$ *if* $p \models \varphi_i$ *for all* $i \in I$.
- $p \models \bigvee_{i\in I} \varphi_i$ *if* $p \models \varphi_i$ *for some* $i \in I$.
- $p \models \langle a \rangle\varphi$ *if there exists some p' such that $p \xrightarrow{a} p'$ and $p' \models \varphi$.*
- $p \models [a]\varphi$ *if $p' \models \varphi$ for all p' such that $p \xrightarrow{a} p'$.*

Let $\mathcal{S}_{CC}(p)$ denote the class of covariant-contravariant simulation formulas satisfied by the process p, that is, $\mathcal{S}_{CC}(p) = \{\varphi \in \mathcal{L}_{CC} \mid p \models \varphi\}$. We will write $p \preceq_{CC} q$ if $\mathcal{S}_{CC}(p) \subseteq \mathcal{S}_{CC}(q)$.

The case of input/output transition systems is probably the clearest example where the covariant-contravariant duality must be applied in order to capture the appropriate simulation order. Input actions should have a covariant behavior reflecting the fact that a reactive system is expected to be "better" whenever it accepts a maximal set of requests; as a consequence, its logical characterization can only capture liveness properties. Conversely, output actions should be contravariant: whenever we specify a system we expect to control its behavior as much as possible, and outputs are generative, which means not controllable by the user. This contravariant character is captured by the universal operator $[a]$, which is only able to define safety properties.

Therefore, the logic $\mathcal{L}_{CC}$ includes formulas that simultaneously capture liveness and safety at a local level, depending on the character of the actions that are used. This is not enough to adequately state all the requirements one could possible need: certainly, after developing a myriad of different semantics for processes [13,5], we would not expect that just by fiddling with one of the simplest, the simulation semantics, we would have the definite answer to treat together covariant and contravariant actions. We are also investigating the covariant-contravariant version of other semantics but, in order to establish which are the basic facts to take into account, it is clear to us that the case of plain simulation is definitely a basic keystone.

Proposition 1. $p \lesssim_{CC} q \iff p \preceq_{CC} q$.

Proof. We will first prove the implication from left to right. Assume that we have pSq for some covariant-contravariant simulation S: we must show that for each $\varphi \in \mathcal{L}_{CC}$, $p \models \varphi$ implies $q \models \varphi$. We proceed by structural induction over φ.

- $q \models \mathsf{tt}$, trivially.
- Let $p \models \langle a \rangle \varphi$ with $a \in A^r \cup A^{bi}$. Then there is p' such that $p \xrightarrow{a} p'$ with $p' \models \varphi$. Now, since pRq and $a \in A^r \cup A^{bi}$ there must be a q' such that $q \xrightarrow{a} q'$ with $p'Rq'$ and, by induction hypothesis, $q' \models \varphi$, that is, $q \models \langle a \rangle \varphi$.
- Let $p \models [a]\varphi$. Then for all p' such that $p \xrightarrow{a} p'$ we have $p' \models \varphi$. Let q' be such that $q \xrightarrow{a} q'$ then, since pSq and $a \in A^l \cup A^{bi}$, there exists p' such that $p \xrightarrow{a} p'$ and $p'Sq'$. By induction hypothesis, since $p' \models \varphi$ then $q' \models \varphi$, that is, $q \models [a]\varphi$.
- Let $p \models \bigwedge_{i \in I} \varphi_i$. Then $p \models \varphi_i$ for all $i \in I$, so by induction hypothesis $q \models \varphi_i$ for all $i \in I$ and then $q \models \bigwedge_{i \in I} \varphi_i$.
- $p \models \bigvee_{i \in I} \varphi_i$. It is analogous to the previous case.

For the other implication let us assume that $p \preceq_{CC} q$ and show that $\preceq_{CC}$ is a covariant-contravariant simulation. Let $a \in A^r \cup A^{bi}$ and $p \xrightarrow{a} p'$; then there exists q' such that $q \xrightarrow{a} q'$ and $p' \preceq_{CC} q'$. Otherwise, we have that for all $q \xrightarrow{a} q'$, $p' \not\preceq_{CC} q'$, that is, we have formulas $\varphi_{q'}$ such that $\varphi_{q'} \in \mathcal{S}_{CC}(p') \setminus \mathcal{S}_{CC}(q')$. Now, taking $\phi = \langle a \rangle \bigwedge_{q'} \varphi_{q'}$, we have $p \models \phi$ and, by hypothesis, also $q \models \phi$. That means that there exists some q_0' such that $q \xrightarrow{a} q_0'$ with $q_0' \models \bigwedge_{q'} \varphi_q'$. But this cannot be the case since $q_0' \not\models \varphi_{q_0'}$.

Now let $a \in A^l \cup A^{bi}$ and $q \xrightarrow{a} q'$; similarly we must show that there exists p' such that $p \xrightarrow{a} p'$ and $p' \preceq_{CC} q'$. By way of contradiction, if for all $p \xrightarrow{a} p'$ we have $p' \not\preceq_{CC} q'$, there are formulas $\varphi_{p'} \in \mathcal{S}_{CC}(p') \setminus \mathcal{S}_{CC}(q')$. Taking $\phi = [a] \bigvee_{p'} \varphi_{p'}$ we have $p \models \phi$ and then by hypothesis $q \not\models \phi$, but this cannot be since $q' \not\models \varphi_{p'}$ for all p'. □

3.2 Conformance Simulations

Conformance simulation can be considered to be a variant of the covariant-contravariant framework in which, instead of separating the actions in several classes, we have a mixed uniform behavior for all the actions. This is brought forward by the fact that if a process cannot execute a, then $p \lesssim_{CS} p + aq$. However, once we have $a \in I(p)$ the contravariant character shows since then $p + aq \lesssim_{CS} p$.

This mixed character of all the actions is now captured at the logical level by a new modal operator a, whose semantics is defined by: $p \models a\varphi$ *if* $p \xrightarrow{a}$ *and* $p' \models \varphi$ *for all* $p \xrightarrow{a} p'$. It is quite interesting to observe that we can alternatively define a as "$\langle a \rangle \wedge [a]$", since we have: $p \models a\varphi \iff p \models \langle a \rangle \varphi$ *and* $p \models [a]\varphi$, which also reveals the mixed intended nature of all the actions in the conformance framework.

Definition 4. *The class $\mathcal{L}_{CS}$ of conformance simulation formulas over A is defined recursively by:*

- $\mathsf{tt} \in \mathcal{L}_{CS}$.
- *If I is a set and $\varphi_i \in \mathcal{L}_{CS}$ for all $i \in I$ then $\bigwedge_{i \in I} \varphi_i, \in \mathcal{L}_{CS}$, $\bigvee_{i \in I} \varphi_i \in \mathcal{L}_{CS}$.*
- *If $\varphi \in \mathcal{L}_{CS}$ and $a \in A$ then $a\varphi \in \mathcal{L}_{CS}$.*

The corresponding satisfaction relation $\models$ *is defined recursively by:*

- $p \models \mathsf{tt}$.
- $p \models \bigwedge_{i \in I} \varphi_i$ *if* $p \models \varphi_i$ *for all* $i \in I$.
- $p \models \bigvee_{i \in I} \varphi_i$ *if* $p \models \varphi_i$ *for some* $i \in I$.
- $p \models a\varphi$ *if* $p \xrightarrow{a}$ *and* $p' \models \varphi$ *for all* $p \xrightarrow{a} p'$.

Let $\mathcal{S}_{CS}(p)$ *denote the class of conformance simulation formulas satisfied by the process* p*, that is,* $\mathcal{S}_{CS}(p) = \{\varphi \in \mathcal{L}_{CS} \mid p \models \varphi\}$*. We will write* $p \preceq_{CS} q$ *if* $\mathcal{S}_{CS}(p) \subseteq \mathcal{S}_{CS}(q)$*.*

One now expects that the liveness and safety requirements will be captured simultaneously and this is indeed the case since from $p \models a\varphi$ we know both that p is able to execute a and that, after executing it in any possible way, φ will be satisfied. Therefore, conformance simulation proves to be quite a reasonable semantics whenever we do not want to distinguish between reactive and generative actions, as discussed in the previous section.

Proposition 2. $p \lesssim_{CS} q \iff p \preceq_{CS} q$.

Proof. We first prove the implication from left to right. Assume that we have pRq for some conformance simulation R: we must show that for each $\varphi \in \mathcal{L}_{CS}$, $p \models \varphi$ implies $q \models \varphi$. The proof will follow by structural induction over φ, the case for tt being trivial.

- Let $p \models a\varphi$. Then, for all $p \xrightarrow{a} p'$ we have $p' \models \varphi$ and there exists at least one such p'. Since pRq also $q \xrightarrow{a}$, and it remains to prove that $q' \models \varphi$ for all successors $q \xrightarrow{a} q'$. Let q'_0 be such that $q \xrightarrow{a} q'_0$. Again, since pRq and $p \xrightarrow{a}$, for each $q \xrightarrow{a} q'$ there exists some $p \xrightarrow{a} p'$ such that $p'Rq'$. So, for q'_0 there exists p'_0 such that $p'_0Rq'_0$ and, since $p'_0 \models \varphi$, by induction hypothesis also $q'_0 \models \varphi$. Thus $q \models a\varphi$.
- Let $p \models \bigwedge_{i \in I} \varphi_i$. Then $p \models \varphi_i$ for all $i \in I$, so by induction hypothesis $q \models \varphi_i$ for all $i \in I$ and then $q \models \bigwedge_{i \in I} \varphi_i$.
- $p \models \bigvee_{i \in I} \varphi_i$. It is analogous to the previous case.

For the other implication, let us assume that $p \preceq_{CS} q$: we show that $\preceq_{CS}$ is a conformance simulation. First, if $p \xrightarrow{a}$ then, since $\mathcal{S}_{CS}(p) \subseteq \mathcal{S}_{CS}(q)$ and $p \models a\mathsf{tt}$, also $q \models a\mathsf{tt}$ and hence $q \xrightarrow{a}$. Now, let $q \xrightarrow{a} q'$ and $p \xrightarrow{a}$. Let us see that there exists some p' such that $p \xrightarrow{a} p'$ and $p' \preceq_{CS} q'$. By way of contradiction, if $p' \not\preceq_{CS} q'$ for all such p', then for each p' there is a formula $\varphi_{p'} \in \mathcal{S}_{CS}(p') \setminus \mathcal{S}_{CS}(q')$. Let $\phi = a \bigvee_{p'} \varphi_{p'}$. It is easy to see that $p \models \phi$: indeed, for each p' such that $p \xrightarrow{a} p'$, $p' \models \varphi_{p'}$. Since $p \preceq_{CS} q$, it must also be the case that $q \models \phi$, that is, for each q'' such that $q \xrightarrow{a} q''$, $q'' \models \bigvee_{p'} \varphi_{p'}$; but $q \xrightarrow{a} q'$ and $q' \not\models \varphi_{p'}$ for any p', contradicting the fact that $q \models \phi$. ☐

4 Some Examples and a Short Discussion

We will start by illustrating the behavior of covariant-contravariant simulations in the case in which we distinguish between input (reactive) and output (generative) actions. Consider the following expending machines:

$$\begin{array}{lll} \mathsf{onecoke} & : & \mathsf{coin} \rightarrow \mathsf{coke} \rightarrow 0 \\ \mathsf{cokeorlemonade} & : & \mathsf{coin} \rightarrow \big((\mathsf{coke} \rightarrow 0) + (\mathsf{lemonade} \rightarrow 0)\big) \end{array}$$

The classical approach would consider $\mathsf{onecoke} \lesssim_S \mathsf{cokeorlemonade}$. However, if the drinks are provided by the machine in an autonomous way then they should be formalized as outputs, which leads us to

$$\mathsf{cokeorlemonade} \lesssim_{CC} \mathsf{onecoke}.$$

This is justified by the fact that choices between generative actions become internal and therefore generate (undesired) non-deterministic behavior.

At the logical level the difference between the two processes above can be brought forward by means of the formula $\langle\mathsf{coin}\rangle\ [\mathsf{lemonade}]\ \mathsf{ff}$, which $\mathsf{onecoke}$ satisfies but $\mathsf{cokeorlemonade}$ does not. It could be thought that the process $\mathsf{cokeorlemonade}$ is being punished for offering lemonade besides coke, but this would be an incorrect interpretation because it follows the classical reactive approach where simultaneous offers mean "the user makes his choice"; instead, when outputs are generative it is the machine that chooses. As a consequence, from $\mathsf{cokeorlemonade} \not\models \langle\mathsf{coin}\rangle[\mathsf{lemonade}]\ \mathsf{ff}$ we implicitly infer that it could be the case that after inserting a coin we did not get our favorite drink (Coke).

Let us now show the differences between covariant-contravariant and conformance simulations. First, at the formal level, the fact that the modal operator a can be defined as "$\langle a\rangle \wedge [a]$" does not mean that these two basic modal operators can appear separately in a formula characterizing $\lesssim_{CS}$. Obviously this cannot be the case since separated $\langle a\rangle$ operators characterize plain simulation, and for the process $\mathsf{choice_coke_lemonade}$: $(\mathsf{coin} \rightarrow \mathsf{coke} \rightarrow 0) + (\mathsf{coin} \rightarrow \mathsf{lemonade} \rightarrow 0)$ we have

$$\mathsf{choice_coke_lemonade} \models \langle\mathsf{coin}\rangle\langle\mathsf{lemonade}\rangle\mathsf{tt} \qquad \mathsf{onecoke} \not\models \langle\mathsf{coin}\rangle\langle\mathsf{lemonade}\rangle\mathsf{tt}$$

but $\mathsf{choice_coke_lemonade} \lesssim_{CS} \mathsf{onecoke}$.

Now, if we consider de universal operator $[a]$, its weakness when used alone arises when it is trivially satisfied. For instance, we have $0 \models [\mathsf{coin}]\ \mathsf{ff}$ but $\mathsf{onecoke} \not\models [\mathsf{coin}]\ \mathsf{ff}$ and $0 \lesssim_{CS} \mathsf{onecoke}$.

One could infer that conformance simulation is the definitive solution to capture all the natural requirements in an specification. Certainly, it combines covariant and contravariant aspects in a very balanced way, but the fact that it treats all the actions uniformly makes it impossible to capture the difference between input and output actions. In particular: $\mathsf{onecoke} \lesssim_{CS} \mathsf{cokeorlemonade}$ but we have already discussed that when outputs are generative, choices always generate non-deterministic behaviors that $\lesssim_{CS}$ is not punishing at all.

On the other hand, choices between equal actions are also considered "harmful" by the conformance semantics so that if $p \lesssim_{CS} q$ then $ap =_{CS} ap + aq$. This is sometimes a too pessimistic approach, which we can illustrate by the following slot_machine specification:

$$\mathsf{slot_machine} : (\mathsf{coin} \rightarrow \mathsf{souvenir} \rightarrow 0) + (\mathsf{coin} \rightarrow ((\mathsf{million\$} \rightarrow 0) + (\mathsf{souvenir} \rightarrow 0)))$$

which becomes conformance simulation equivalent to the pluff_machine

$$\mathsf{pluff_machine} : \mathsf{coin} \rightarrow \mathsf{souvenir} \rightarrow 0$$

In this case the possible return of the big pot is not taken into account at all. Obviously, the solution comes from choosing in each case the adequate semantics to capture accurately the desired behaviors. The bad news is that we need to study many diferent semantics; the good news for us is... the same!, since we are already working on them

References

1. Antonik, A., Huth, M., Larsen, K., Nyman, U., Wasowski, A.: 20 Years of Mixed and Modal Specifications. Bulletin of the European Association for Theor. Comput. Sci. (May 2008)
2. Benes, N., Kretínský, J., Larsen, K.G., Srba, J.: On determinism in modal transition systems. Theor. Comput. Sci. 410(41), 4026–4043 (2009)
3. Cîrstea, C.: A modular approach to defining and characterising notions of simulation. Inf. Comput. 204(4), 469–502 (2006)
4. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC / SIGSOFT FSE, pp. 109–120 (2001)
5. de Frutos Escrig, D., Gregorio-Rodríguez, C., Palomino, M.: On the unification of semantics for processes: observational semantics. In: Nielsen, M., Kucera, A., Miltersen, P.B., Palamidessi, C., Tuma, P., Valencia, F.D. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 279–290. Springer, Heidelberg (2009)
6. de Frutos-Escrig, D., Rosa Velardo, F., Gregorio-Rodríguez, C.: New bisimulation semantics for distributed systems. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 143–159. Springer, Heidelberg (2007)
7. Fábregas, I., de Frutos-Escrig, D., Palomino, M.: Non-strongly stable orders also define interesting simulation relations. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 221–235. Springer, Heidelberg (2009)
8. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM 32(1), 137–161 (1985)
9. Hughes, J., Jacobs, B.: Simulations in coalgebra. Theor. Comput. Sci. 327(1-2), 71–108 (2004)
10. Leduc, G.: A framework based on implementation relations for implementing LOTOS specifications. Computer Networks and ISDN Systems 25(1), 23–41 (1992)
11. Lynch, N.: I/O automata: A model for discrete event systems. In: 22nd Annual Conference on Information Sciences and Systems, pp. 29–38 (1988)
12. Park, D.: Concurrency and automata on infinite sequences. In: GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
13. van Glabbeek, R.J.: The linear time-branching time spectrum I: The semantics of concrete, sequential processes. In: Handbook of process algebra, pp. 3–99. North-Holland, Amsterdam (2001)

Author Index

Balasingham, Ilangko 95
Basu, Ananda 32
Bensalem, Saddek 32
Bocchi, Laura 87
Bokor, Péter 216
Boronat, Artur 2, 47
Bozga, Marius 32
Bruni, Roberto 2
Bu, Lei 155

Caillaud, Benoît 32
Cimatti, Alessandro 155
Crespo, F. Javier 140

de la Encina, Alberto 140
Delahaye, Benoît 32
Dingel, Juergen 125

Fábregas, Ignacio 224
Ferrara, Pietro 186
Frutos Escrig, David de 224

Herrmann, Peter 17
Hierons, Robert M. 63
Holzmann, Gerard J. 1

Jacobs, Bart 170
Johnsen, Einar Broch 95

Kazemeyni, Fatemeh 95
Kraemer, Frank Alexander 17

Legay, Axel 32
Li, Xuandong 155
Llana, Luis 140
Lluch Lafuente, Alberto 2

Meseguer, José 47
Montanari, Ugo 2
Mover, Sergio 155

Núñez, Manuel 63, 78

Ölveczky, Peter Csaba 47
Owe, Olaf 95

Palomino, Miguel 224
Paolillo, Generoso 2
Pardo, Juan José 78
Permpoontanalarp, Yongyuth 201
Piessens, Frank 170
Posse, Ernesto 125

Ruiz, M. Carmen 78

Serafini, Marco 216
Smans, Jan 170
Suri, Neeraj 216

Tonetta, Stefano 155
Tuosto, Emilio 87

Wijs, Anton 110